MINLOG Tutorial

Martin Ruckert

April 30, 1999

1 Introduction

Every (good) system can be used in many different ways, including ways that were not intended or even foreseen by the authors of the system. It is, however, a good start to learn how to use a system as intended by the authors. If you find new, interesting, more convenient ways to use the MINLOG systems, please let us know.
Here, we describe step by step how to use the MINLOG system, progressing from simple to more and more complex tasks. Many details will be omitted. You will find some of these details in the manual [2] and even more details in the source code [3]. For a more high level description of the system, read the papers that are advertised on our web page[4].

2 The Basics

2.1 Emacs

MINLOG is conveniently used with emacs. To start emacs type `emacs` on the command line. emacs was invented before the advent of modern GUI's and hence, you need some time and patience to get used to it. After this, it is a very powerful tool. Emacs, while a bit cryptic to use, is largely self-documenting. After having started emacs, type `Ctrl-h t` (control h followed by t) to start the emacs tutorial. Work through it (approximately 30min.) or you will regret it (as I did).

2.2 Scheme and MINLOG

MINLOG is written in the programming language scheme. The "favorite" dialect of scheme for running MINLOG is LMU-Scheme, a scheme implementation by O. Forster[1].
To get scheme running inside of emacs type `Meta-x run-lmuscheme`. [1]
After LMU-scheme has started, it gives you a "`==>`" prompt and next, you want to start MINLOG. To do so you type `(load "/home/cip02/mkbt1aa/kurs/minlog/src/initlmu")`. But wait—the string here must be the filename including the pathname of the init.scm (or initlmu.scm) file and you might ask whether you are required to type this horrible long filename all the time. No, you are not. Emacs will help you.

[1] If you worked through the emacs tutorial, you should know what the Meta-key is, otherwise, try the Escape-Key or one of the Alt-keys or, on a SUN, the key next to the space bar with the little diamond on it.

The preferred way of working with MINLOG under emacs is in a split window, where you have a text file in one part of the window and scheme running in the second part of the window. You do not type your commands directly into the scheme window, but you type them into a text file where you can edit them, and save them for later use.
To split the window, you type `Ctrl-x 2`. Now you have two windows and one of them should be converted to a text window. Do this by typing `Ctrl-x Ctrl-f` and then type the file name. Don't worry if you do not have a file yet. If the file doesn't exist, it will be created for you. So for instance type `test.scm`. You should use the extension `scm` to make emacs understand that this is a file with scheme code that can serve as input to the scheme process running already.
Now type the line `(load "/home/cip02/mkbt1aa/kurs/minlog/src/initlmu")` in the new window. You can execute this line by placing the cursor at the end of this line (after the closing parenthesis) where it should be conveniently located already after typing the line, and type `Ctrl-x Ctrl-e`.
Scheme will execute the line and will show the output of the command in the scheme window. If you proceed like this, you will be able to save your work later (to save a text file just type `Ctrl-x Ctrl-s`).
Now MINLOG is ready to process your commands. You will type commands in your text window, edit them there, and execute them with `Ctrl-x Ctrl-e`. Before I forget to mention it, you can exit emacs by typing `Ctrl-x Ctrl-c`.

3 Interactive Theorem Proving Using MINLOG

To start with, we look at a simple example in propositional logic.

3.1 Propositional logic

3.1.1 Entering the Formula

For all propositions $p1$ and $p2$, the formula $((p1 \to p2) \to p1) \to p1$ is true.
Lets try to prove this. First, we have to tell MINLOG about this formula. Type:
`(define peirce-formula (pf "((p1 -> p2) -> p1) -> p1"))`
and execute it. It defines a new variable `peirce-formula` and sets its value to the next item in the list. The `(pf " ...")` needs some more explaining. MINLOG stores formulas internally as special scheme terms and the function `pf`, short for "parse formula" is a convenient way to create these terms. The function `pf` takes a string as an argument describing the formula in a human readable form and returns the internal form of this formula. This will become the value of the new variable `peirce-formula`. It is further important that the propositional variables `p1` and `p2` are formed by appending a number to the letter p. All variables in MINLOG must be declared and they have a type. MINLOG can then determine the type of a variable from its name. For convenience, there are some predefined variable names, as for example "p". It adds to MINLOGs convenience that without a new declaration, new variables can be derived from existing variables by appending a number—hence `p1` and `p2`
After this, you can type `peirce-formula` and scheme will output the value of this variable, that is the internal form of our formula. You will be surprised, it is still quite readable.

3.1.2 Setting a Goal

The first step in a proof is, to tell MINLOG what we want to prove. This is done with the `set-goal` command. You type `(set-goal '? peirce-formula)`.
MINLOG keeps a list of goals. Goals are formulas that must be proved. Typically, a logical rule reduces the proof of a formula to the proof of one or more other formulas, called the premises of the rule. And therefore MINLOG must be able to handle not just one goal but a whole list of goals. To be able to identify each goal, the goals are named, and as a matter of convention, the names of goals start with a question mark. The top goal, as you see above, has simply the name "?".

3.1.3 Planning the Proof

Now, we need to do some mathematics. We have to make an informal plan on how to prove this formula, that is, we have to convince ourself that this formula is definitely true.

Very informal Argument: Consider the statement "If Monica Lewinsky meets Pierre Fermat, then they talk about number theory". Is it true? Yes, it is, because we can assume that the meeting never took place and therefore the implication statement is true (that's how classical logic works). Now consider the statement "If Monica Lewinsky meets Bill Clinton, then they talk about politics". Whether this is true is not so clear. However, a spokesperson of the white house assures us: "If this statement is true then you may conclude that Monica and Bill did meet indeed". Now did they meet? Yes, they did!
If they **did not meet**, we know that the implication is true, as it was in Fermat's case, and believing the white house spokesperson, we must then conclude that they **did meet**. This is a contradiction and since the world of politics is without contradictions, we must conclude that they did meet, however, we still do not know whether they talked about politics (what a clever spokesperson).

Detailed (still Informal) Argument: Let's assume that $(p1 \to p2) \to p1$ is true, then we have to show $p1$ (our goal). We prove $p1$ indirectly, by assuming $p1$ is false and deriving a contradiction. If $p1$ is false, then $p1 \to p2$ will be true.
To proof $p1 \to p2$, we assume $p1$ and have to prove $p2$. Now, the assumption $p1$ is a contradiction to our previous assumption that $p1$ is false. There is a logical rule, called "ex falso quodlibet" (I hope you know Latin) that from a contradiction everything can be derived—for example $p2$.
Having $p1 \to p2$ and our first assumption $(p1 \to p2) \to p1$ we can conclude $p1$. This again is in contradiction to the assumption that $p1$ is false. From this contradiction, we finally conclude that the assumption "$p1$ is false" was wrong, that is that $p1$ is true (this last argument: "from (x is false) is false follows x" is called "stability"). Now we have derived $p1$ from $(p1 \to p2) \to p1$ and we can conclude that $((p1 \to p2) \to p1) \to p1$.
Note, that the two logical rules used, "ex falso quodlibet" and "stability" are closely related.

3.1.4 Implementing the Proof

Our next step is the complete formalization of the previous argument using the MINLOG system. When formalizing a proof with MINLOG, the most important thing to know is:

MINLOG is designed to do "Goal Driven Reasoning", also called "Backward Chaining". This means, we start by setting up the final conclusion as the top goal and unroll the proof from the back.
Our last step in the above proof was obtaining the final goal using the assumption $(p1 \to p2) \to p1$ Hence our first step in MINLOG is to tell the system about this assumption.
To make an assumption type

```
(assume 'whs)
```

this will take the goal, which is an implication, turns the first part of the implication into an assumption and establishes the second part as the new goal. `whs` is the name of the assumption (whs stands for white house spokesperson) and the quote in front of it prevents scheme to interpret `whs` as a scheme variable (to be replaced by its value).
MINLOG will print

```
;ok, under these assumptions
;we have the new goal
;?-kernel: p1 from
;  whs:(p1 -> p2) -> p1
```

How did our informal proof obtain $p1$? By using stability! Here it becomes obvious that you should make an informal proof first, since only very experienced MINLOGicans will guess that you have to use stability at this point without first constructing an informal argument.
Using stability exemplifies an other proof technique: the use of lemmata or "Global Assumptions" as they are called in MINLOG. Stability for predicates is a predefined global assumption.
So type

```
(use-with 'stab-log '?nnm)
```

Here `stab-log`, for "stability", is the name of the global assumption to be applied to the top goal and `?nnm` becomes the name of the new top goal. Since it is a goal and we will obtain the double negation of $p1$ (meeting), we use the name `?nnm`.
MINLOG will print

```
;ok, ?-kernel can be obtained from
;?nnm: (p1 -> bot) -> bot from
;  whs:(p1 -> p2) -> p1
```

showing the new goal $(p1 \to \bot) \to \bot$.
The "$\to \bot$" (the symbol $\bot$ is called "bottom" and means "false") is a different way of expressing "not" or "is false", therefore the new goal reads "($p1$ is false) is false" as expected.
This new goal was obtained in the informal proof by means of the assumption "$p1$" is false" and this step is reflected in MINLOG by the command

```
(assume 'nm)
```

Here "nm" stands for "no meeting". The command will produce

```
;ok, under these assumptions
;we have the new goal
;?nnm-kernel: bot from
;  whs:(p1 -> p2) -> p1
;  nm:p1 -> bot
```

showing that MINLOG now expects us to prove a contradiction $\bot$. This contradiction was derived from the assumption $p1 \to \bot$ and a proof of $p1$.
Similar to the step above, where we used the global assumption `stab-log`, now we use the (local) assumption `nm`. Again, the command needs the name of the new goal $p1$. We type

```
(use-with 'nm '?m)
```

using `?m` (for "meeting") as the name for the goal $p1$ and get

```
;ok, ?nnm-kernel can be obtained from
;?m: p1 from
;  whs:(p1 -> p2) -> p1
;  nm:p1 -> bot
```

Informally, we had proved $p1$ from $p1 \to p2$ by using the statement `whs`. This step is reflected again by a `use-with` command with the assumption `whs`. For the name of the new goal $p1 \to p2$, we choose this time `?tp` (talking politics).
After typing

```
(use-with 'whs '?tp)
```

MINLOG prints

```
;ok, ?m can be obtained from
;?tp: p1 -> p2 from
;  whs:(p1 -> p2) -> p1
;  nm:p1 -> bot
```

The next command

```
(assume 'm)
```

removes again the $p1 \to \ldots$ from the goal adding the assumption `m: p1`. We obtain

```
;ok, under these assumptions
;we have the new goal
;?tp-kernel: p2 from
;  whs:(p1 -> p2) -> p1
;  nm:p1 -> F
;  m:p1
```

How did we prove $p2$. There is actually no information what so ever about the proposition $p2$ and the only way to prove it is the "ex falso quodlibet" rule. This rule is again a global assumption. The name of the new goal `?c` stands for "contradiction". Type

```
(use-with `efq-log `?c)
```

and get

```
;ok, ?tp-kernel can be obtained from
;?c: bot from
;  whs:(p1 -> p2) -> p1
;  nm:p1 -> bot
;  m:p1
```

We prove the contradiction by combining the two assumptions `m` and `nm` with the command

```
(use-with 'nm 'm)
```

and finally see the message

```
;ok, ?c is proved. Proof finished.
```

We made it!

3.1.5 Summary of Commands

- `Ctrl-h t` Start the emacs tutorial
- `Meta-x run-lmuscheme` Start LMU scheme
- `(load "/home/cip02/mkbt1aa/kurs/minlog/src/initlmu")` Load MINLOG
- `Ctrl-x 2` Split the emacs window
- `Ctrl-x Ctrl-f` Load a file into emacs
- `Ctrl-x Ctrl-s` Save the file
- `Ctrl-x Ctrl-c` Exit emacs
- `Ctrl-x Ctrl-e` Executing the scheme expression preceding the cursor
- `(define` *scheme-variable value*`)` define a new *scheme variable* and give it a *value*. Example `(define peirce-formula (pf "((p1 -> p2) -> p1) -> p1"))`
- `(pf "`*string*`")` parse the given *string* to convert a MINLOG formula from external form into internal form. Example:`(pf "((p1 -> p2) -> p1) -> p1")`
- `(assume` *new-assumption*`)` Applies if the current top goal is an implication. The left hand side of the implication becomes a new assumption with name "*new assumption*" and the right hand side becomes the new top goal. Example: `(assume 'whs)`
- `(use-with` *formula new-goal*`)` Applies the given *formula* to the top goal. The *formula* might be a global assumption—a lemma which was proved earlier or is predefined—, or a local assumption. It is followed by the name of the *new goal*. Instead of the new goal also the name of an existing assumption can be supplied. Example: `(use-with 'whs '?tp)`

  The `use-with` command is actually much more powerful, and we will complete its description in the command summarys of the next sections.

3.2 Quantifiers

We start by looking at the following example: $\exists n(r(n) \to \forall k r(k))$.
This formula has many practical applications, for example, it is commonly used to produce weather forecasts. Applied to the typical Munich summer weather, this formula says: there is a day during summer, and if it rains on this day, then it will rain all summer long.

3.2.1 Informal Proof

In Munich, this statement is often true simply because it rains all summer long. Let's however assume that, may be this summer, we will have at least one warm and sunny day, and let's plan a picnic for this day. In this case, could the formula possibly be false? No! Because the statement "if it rains on our picnic day (false), then it will rain all summer long (again false)" is a true statement, and our picnic day is just the day that the formula claimed would exist.
Before we will use MINLOG to do a rigorous formal proof, we should try to further formalize this proof. Our previous considerations suggests an indirect approach. We assume the formula is false and derive a contradiction.
If the formula $\exists n(r(n) \to \forall k r(k))$ is false, we have: $\forall n((r(n) \to \forall k r(k)) \to \bot)$ which we shall call the "no-forecast" formula.
Let's assume we have a nice day m for our picnic. We first prove, again indirectly, that this day (you can guess it) is necessarily a rainy day.
Assume, until we know better, that we have chosen a sunny day for our picnic: $r(m) \to \bot$. Knowing that $r(m)$ is false, we can derive from it $\forall k r(k)$ using the rule "ex falso quodlibet" and have $r(m) \to \forall k r(k)$. Now, the no-forecast formula, applied to our picnic day, just says that this is false. We have a contradiction. Knowing, that the assumption of a picnic day without rain is false, we can conclude, using the rule of "stability" that it rains on the picnic day.
The above proves the well known fact, that regardless how you choose your picnic day, it will rain on this day. We can summarize this result, introducing an $\forall$-quantifier: $\forall m r(m)$. We can weaken this sad result by the additional premise $r(n)$ and obtain $r(n) \to \forall m r(m)$. We apply again the no-forecast formula to arrive at a contradiction. This tells us that the no-forecast formula was false after all, and we can finally conclude $\exists n(r(n) \to \forall m r(m))$. Unfortunately, this proof is not a "constructive" proof. It does not provide any clue about how to choose, in advance, the right day for the picnic.

3.2.2 Implementing the Proof

Before we can enter the formula and set the goal for MINLOG, we tell MINLOG about the new predicate `rain`. This is a unary predicate taking a natural number as argument and has a boolean value. Such a predicate can be defined with the command

```
(add-predicate-constant 'rain (c-arity 'nat))
```

The function `c-arity` takes a sequence of types (here `nat` for natural numbers), and constructs an arity expression from it. This arity defines the signature of the new predicate constant `rain`.
Now, we can proceed:

```
(define weather-formula (pf "excl n . rain n -> all m rain m"))
```

The formula $\exists n(r(n) \to \forall m r(m))$ is again created from a string using the "parse formula" function `pf`. We can see the two quantifiers `all` and `excl`. The `excl` is the existential quantifier of classical logic which is different form the existential quantifier of constructive logic which we will encounter in the next section. The classical quantifier can be defined by $\neg\forall\neg$ and is available in MINLOG just for notational convenience. Each quantifier is followed by a variable (`n` and `m`) which is bound by the quantifier. By default, the quantifier binds the variable only in the immediately following term. If, however, the variable is followed by a dot ".", then the variable is bound in the maximal possible term. Without using the dot, we could have written `"excl n ( rain n -> all m rain m)"`.
No parenthesis are needed to enclose the arguments of the predicate `rain`. The parameters must, however, agree in type and number with the arity of the predicate. `n` and `m` are predefined variables of type `nat`.
We enter as usual

```
(set-goal '? weather-formula)
```

and MINLOG answers

```
 ;?: excl n.rain n -> all m rain m
```

As mentioned before, internally the classical existential quantifier has the form $\neg\forall\neg$ and again the "$\neg\cdots$" has the form "$\cdots \to \bot$". The internal formula therefore reads: $(\forall n((r(n) \to \forall m r(m)) \to \bot)) \to \bot$. It is an implication stating that a contradiction follows from the "no-forecast" formula, as we called it above. To start the indirect proof, we use

```
(assume 'no-forecast)
```

and, as expected, MINLOG prints

```
;ok, under these assumptions we have the new goal
;?-kernel: bot from
;  no-forecast:all n.(rain n -> all m rain m) -> bot
```

We follow the same strategy used earlier and read our informal proof backwards to see how we were able to prove the contradiction `bot`. We did so by using the no-forecast formula. This formula has the form $\forall n(\cdots \to \bot)$. It is not in the form $\cdots \to \bot$ required to infer our current goal $\bot$. The no-forecast formula needs to be applied to a specific n before it can be used. All necessary steps can be executed by a single `use-with` statement:

```
(use-with 'no-forecast 'n '?forecast-n)
```

It takes the no-forecast formula and applies it to `n` and then applies it to `?forecast-n` creating a new goal and reaching the desired formula `bot`. The `use-with` command is able to handle a whole list of applications as further explained in the command summary below.
MINLOG prints

```
;ok, ?-kernel can be obtained from
;?forecast-n: rain n -> all m rain m from
;  no-forecast:all n.(rain n -> all m rain m) -> bot
;  n
```

We proved the new goal by weakening "`all m rain m`". We achieve this effect by using the `assume` command to make the first part of the implication an assumption, and then use the new command `drop` to throw away this assumption. Type:

```
(assume 'some-rain)
(drop 'some-rain)
```

MINLOG prints:

```
;ok, under these assumptions we have the new goal
;?forecast-n-kernel: all m rain m from
;  no-forecast:all n.(rain n -> all m rain m) -> bot
;  n  some-rain:rain n

;ok, we now have the new goal
;?forecast-n-kernel-dropped: all m rain m from
;  no-forecast:all n.(rain n -> all m rain m) -> bot
```

We proved $\forall m r(m)$ by proving it for the arbitrary picnic day. To do this, we could use one of the predefined variables `i`, `j`, `k`, `l`, `m`, or `n` of type `nat` and apply the current goal to this variable, but we would have missed a good opportunity to demonstrate how to define new variables. So, let's be more verbose and use a variable named, for example, "`picnic`" instead.
We declare the variable with

```
(add-variable 'picnic 'nat)
```

and continue with

```
(assume 'picnic)
```

This results in

```
;ok, under these assumptions we have the new goal
;?forecast-n-kernel-dropped-kernel: rain picnic from
;  no-forecast:all n.(rain n -> all m rain m) -> bot
;  picnic
```

We used stability to conclude indirectly that it rains on the picnic day. Type

```
(use-with 'stab-log 'picnic '?indirect)
```

But why is the additional parameter `picnic` needed? Global assumptions, like logical stability, are always closed formulas, that is, they do not contain free variables. If stability is used to obtain `rain picnic` the formula reads `((rain picnic -> bot) -> bot) -> rain picnic`. It contains the free variable `picnic`. To remedy this, all free variables are automatically bound by a $\forall$ quantifier. Then stability reads "`all picnic .  ((rain`

picnic -> bot) -> bot) -> rain picnic". Before this formula can be used to conclude "rain picnic", it must be applied to the variable picnic—hence the additional argument.
Now the situation is:

```
;ok, ?forecast-n-kernel-dropped-kernel can be obtained from
;?indirect: (rain picnic -> bot) -> bot from
;  no-forecast:all n.(rain n -> all m rain m) -> bot
;  picnic
```

Next we assume that we have a sunny picnic:

```
(assume 'sunny-picnic)
```

and get:

```
;ok, under these assumptions
;we have the new goal
;?indirect-kernel: bot from
;  no-forecast:all n.(rain n -> all m rain m) -> bot
;  picnic  sunny-picnic:rain picnic -> bot
```

To conclude the proof, we use the no-forecast formula for the picnic day to prove bot, then we prove the implication "rain picnic -> all m rain m" by first assuming a "rainy-picnic and deriving all m rain m using the rule "ex falso quodlibet". The falsum is easily obtained from the contradiction sunny-picnic and rainy-picnic. The MINLOG interaction reads:

```
==> (use-with 'no-forecast 'picnic '?picnic-forecast)
;ok, ?indirect-kernel can be obtained from
;?picnic-forecast: rain picnic -> all m rain m from
;  no-forecast:all n.(rain n -> all m rain m) -> bot
;  picnic  sunny-picnic:rain picnic -> bot

==>(assume 'rainy-picnic)
;ok, under these assumptions we have the new goal
;?picnic-forecast-kernel: all m rain m from
;  no-forecast:all n.(rain n -> all m rain m) -> bot
;  picnic  sunny-picnic:rain picnic -> bot
;  rainy-picnic:rain picnic

==> (use-with 'efq-log '?contradiction)
;ok, ?picnic-forecast-kernel can be obtained from
;?contradiction: bot from
;  no-forecast:all n.(rain n -> all m rain m) -> bot
;  picnic  sunny-picnic:rain picnic -> bot
;  rainy-picnic:rain picnic

==> (use-with 'sunny-picnic 'rainy-picnic)
;ok, ?contradiction is proved. Proof finished.
```

3.2.3 Summary of Commands

- (`drop` *assumption*) removes an assumption, that is no longer needed, from the proof. The command is used for purely cosmetic reasons.

  Example: `(drop 'some-rain)`.

- (`add-variable` *variable-name type*) declares a new variable. Additional variables can be used as needed by appending a number to the same base name. The type of the variable is `nat` or `obj` or a higher type as we will see later. Example: `(add-variable 'picnic 'nat)` defines variables `picnic` or `picnic1` or `picnic55` for natural numbers.

- (`add-predicate-constant` *name arity*) defines a new predicate with the given *name* and *arity*. The predicate constant can then be applied to appropriate arguments, as given by *arity*, to obtain a boolean term (or atomic formula). Example:`(add-predicate-constant 'rain (c-arity 'nat))`

- (`c-arity` *typelist*) creates an arity from a sequence of types. It is used to construct arguments for the `add-predicate-constant` function. Example: `(c-arity 'nat 'nat)` is the arity of a predicate with two arguments of type `nat`.

- (`use-with` *formula arguments*) applies the given *formula* to derive the current goal. The *formula* might be a global assumption—a lemma which was proved earlier or is predefined—, or a local assumption. It is followed by an *argumentlist*. The *formula* is first applied to the given arguments and finally used to derive the current goal.

  In the simple case of a single argument, this means that the *formula* must have the form

  - *argument* `->` *current-goal*, if the argument is a local or global assumption or a new goal;

    it must have the form

  - `all` *variable* `->` *current-goal*, if the *argument* is a term such that the substitution of *variable* in *goal* by the *argument* produces the current goal.

  With multiple arguments, this application process is repeated for each argument.

  Assume for example that we have the assumption "`u1 : all n . r n -> all k . p ->r k `" and "`u2 : r 0 `", and the current goal is "`?top : r m`" then we can write

	MINLOG Code	Comment
(	`use-with`	
	`'u1`	$\forall n(r(n) \to \forall k(p \to r(k)))$
	`0`	$r(0) \to \forall k(p \to r(k))$
	`'u2`	$\forall k(p \to r(k))$
	`'m`	$(p \to r(m))$
	`'?newgoal`	the new goal is `?newgoal : p`
)		and we reached $r(m)$, the current goal

Example:`(use-with 'no-forecast 'picnic '?picnic-forecast)`

3.3 Arithmetic

In this section, we will learn about induction. It is the main concept of Arithmetic in regard to proof theory. In addition, we will encounter the "and" operator and the constructive existential quantifier. We will see that using the constructive quantifier will enable us to read our proofs as programs. Our example in this sections is the Fibonacci Numbers.
The Fibonacci Numbers f_i are a series of natural numbers defined by $f_0 = 1$, $f_1 = 1$ and $f_{n+2} = f_n + f_{n+1}$. It is pretty obvious that these numbers are well defined, however, since we do not study Fibonacci Numbers but induction, we will attempt a proof. The proof is far less obvious than you might think.

3.3.1 Informal Proof

Using a binary predicate f meaning that $f(n,k)$ is true if and only if $f_n = k$, we can formulate the theorem like this: Given $f(0,1)$, $f(1,1)$ and $f(n,k) \wedge f(n+1,l) \rightarrow f(n+2,k+l)$ we claim $\forall n \exists k f(n,k)$.
To prove our claim, the natural thing is to use induction by n. The induction is a tick more complicated here since the induction hypothesis for $f(n,k)$ will not allow us to conclude anything about $f(n+2,\ldots)$ since we would need $f(n+1,\ldots)$. The usual trick is to prove a slightly stronger formula by induction. This formula reads: $\forall n(\exists k f(n,k) \wedge \exists l f(n+1,l))$. It is easy to prove the weaker goal from this stronger goal, since it is just the first part of the conjunction.
Now we can start the induction:
Induction Base $n = 0$: We have to prove $\exists k f(0,k) \wedge \exists l f(0+1,l))$. The first part is true for $k = 1$ and the second part for l $l = 1$ according to our assumptions about f.
Induction Step: Let's assume (induction hypothesis) that we have $\exists k f(n,k) \wedge \exists l f(n+1,l)$ then we have to show that $\exists k f(n+1,k) \wedge \exists l f(n+1+1,l)$ is true.
The first part is true because it is the second part of the induction hypothesis. It remains to prove the second part.
Our induction hypothesis promises the existence of a k and a l with $f(n,k)$ and $f(n+1,l)$. We take these two numbers and use the recursive definition of Fibonacci numbers $f(n,k) \wedge f(n+1,l) \rightarrow f(n+2,k+l)$ to conclude that the sum of these two numbers $l+k$ is exactly what we were looking for. From $f(n+2,k+l)$ we conclude $\exists l f(n+1+1,l)$.

3.3.2 Formal Proof

The command

```
(add-predicate-constant 'fib (c-arity 'nat 'nat))
```

needs no further explanation.
Next we type

```
(add-global-assumption 'fib0 (pf "fib 0 1"))
(add-global-assumption 'fib1 (pf "fib 1 1"))
(add-global-assumption 'fibn
    (pf "all n,k,l.fib n k & fib(n+1)l -> fib(n+2)(k+l)"))
```

The command "add-global-assumption" can be abbreviated "aga". It adds the given formula as a global assumption. This way we can use the properties defining the Fibonacci numbers in our proof without having to deal with too many local assumptions.
Next we have:

```
(set-goal '? (pf "all n ex k fib n k"))
```

Now, we do not want to proof this goal but the stronger goal $\forall n(\exists k f(n,k) \wedge \exists l f(n+1,l))$. We introduce this new goal using the command "cut". Note the symbol "&"; it is the and-operator.

```
(cut (pf "all n . ex k fib n k & ex l fib(n+1)l"))
```

Cut will just establish a new goal. After this you have to prove two things: First that your old goal follows from the new goal and second the new goal itself. To make this clear, MINLOG prints

```
;ok, ? can be obtained from
;?-side-premise: all n.ex k fib n k & ex l fib(n+1)l
;?-main-premise: (all n.ex k fib n k & ex l fib(n+1)l) ->
                 all n ex k fib n k
```

Two "assume" commands come next. First, we assume the stronger formula we want to prove by induction and second, we assume the number n. For convenience, several consecutive "assume" commands can be combined into a single one.
In the "use-with" command, which then proves the goal straight forward from the assumption, the "left" needs some explanation: We know already that the "use-with" command applies the assumption to a sequence of arguments. Implications are applied to formulas, $\forall$-quantifiers are applied to terms, and similarly a conjunction can be applied to the symbols "left" or "right" to yield the left or right part of the conjunction. The interaction reads:

```
==> (assume 'stronger 'n)

;ok, under these assumptions we have the new goal
;?-main-premise-kernel: ex k fib n k from
;  stronger:all n.ex k fib n k & ex l fib(n+1)l
;  n

==> (use-with 'stronger 'n 'left)

;ok, ?-main-premise-kernel is proved. The active goal now is
;?-side-premise: all n.ex k fib n k & ex l fib(n+1)l
```

Next comes the induction. In MINLOG you type "(ind)":

```
==> (ind)

;ok, ?-side-premise can be obtained from
;?-side-premise-step: all n.ex k fib n k & ex l fib(n+1)l ->
                              ex k fib(n+1)k & ex l fib(n+1+1)l
;?-side-premise-base: ex k fib 0 k & ex l fib(0+1)l
```

The base case is proved first. The formula is a conjunction and it is proved by proving each part separately. The command "`split`" is used to split the conjunction into two separate goals:

```
==> (split)

;ok, we have the new goals
;?-side-premise-base-right: ex l fib(0+1)l
;?-side-premise-base-left: ex k fib 0 k
```

Notice, that by now we have tree goals waiting to be solved: side-premise-step, side-premise-base-right, and side-premise-base-left. Unlike average mathematicians, MINLOG will never forget about even the most tiny subgoals and will demand a proof sooner or later.
The goal "`ex k fib 0 k`" requires the proof of an (constructive!) existential quantifier. This proof can only be done by providing the object whose existence is claimed. The command to prove an existential statement is "`ex-intro`" and it will need the object that it claims will exist as an argument. Luckily, we know the object (the 1) and we can prove this by using our global assumptions `fib0` and `fib1`.

```
==> (ex-intro (pt "1"))

;ok, ?-side-premise-base-left can be obtained from
;?-side-premise-base-left-main: fib 0 1

==> (use 'fib0)

;ok, ?-side-premise-base-left-main is proved. The active goal now is
;?-side-premise-base-right: ex l fib(0+1)l

==> (ex-intro (pt "1"))

;ok, ?-side-premise-base-right can be obtained from
;?-side-premise-base-right-main: fib(0+1)1

==> (use 'fib1)

;ok, ?-side-premise-base-right-main is proved. The active goal now is
;?-side-premise-step: all n.ex k fib n k & ex l fib(n+1)l ->
                           ex k fib(n+1)k & ex l fib(n+1+1)l
```

Note the `pt` command. It is analog to the `pf` command. `pt` parses terms as `pf` parses formulas.
It remains to prove the induction step. As you might guess, we need an assume statement first (`ih` = induction hypothesis).

```
==> (assume 'n 'ih)
```

```
;ok, under these assumptions we have the new goal
;?-side-premise-step-kernel: ex k fib(n+1)k & ex l fib(n+1+1)l from
;  n  ih:ex k fib n k & ex l fib(n+1)l
```

We use `split` to obtain two separate goals for the conjunction. The first subgoal is trivial proved by the right part of the induction hypothesis.

```
==> (split)

;ok, we have the new goals
;?-side-premise-step-kernel-right: ex l fib(n+1+1)l from
;  n  ih:ex k fib n k & ex l fib(n+1)l

;?-side-premise-step-kernel-left: ex k fib(n+1)k from
;  n  ih:ex k fib n k & ex l fib(n+1)l

==> (use-with 'ih 'right)

;ok, ?-side-premise-step-kernel-left is proved. The active goal now is
;?-side-premise-step-kernel-right: ex l fib(n+1+1)l from
;  n  ih:ex k fib n k & ex l fib(n+1)l
```

It looks like we should be able to prove the remaining goal from the global assumption `fibn` (recursive definition of Fibonacci Numbers) and the induction hypothesis. The crucial point is, that we have to extract the correct numbers l and k from the induction hypothesis first. Only afterwards we can proceed with an `ex-intro` command.
To start with, we need a commands that splits an assumption $A \wedge B$ into two assumptions A and B without affecting the goal. This command is `split-hyp` (split hypothesis). It needs an obvious argument, the name of the assumption. After that we can throw away the old induction hypothesis. Note that while MINLOG typicaly does goal driven reasoning, the `split-hyp` command is an instance of data driven reasoning.

```
==> (split-hyp 'ih)

;ok, the new goal is
;?-side-premise-step-kernel-right-splithyp: ex l fib(n+1+1)l from
;  n  ih:ex k fib n k & ex l fib(n+1)l
;  ih1:ex k fib n k
;  ih2:ex l fib(n+1)l

==> (drop 'ih)

;ok, we now have the new goal
;?-side-premise-step-kernel-right-splithyp-dropped: ex l fib(n+1+1)l from
;  n  ih1:ex k fib n k
;  ih2:ex l fib(n+1)l
```

The command to extract a term from an assumption in existential form is `ex-elim`. Again, the name of the assumption in question is a required argument. This command works by adding a new premise to the current goal, which is the existential formula with the ∃-quantifier replaced by a ∀-quantifier. After this move, we can `drop` the assumption and use `assume` to move the `k` and the formula `fib n k` back to the list of assumptions. Thus we have:

```
==> (ex-elim 'ih1)

;ok,
;?-side-premise-step-kernel-right-splithyp-dropped can be obtained from
;?-side-premise-step-kernel-right-splithyp-dropped-side:
;                                  all k.fib n k -> ex l fib(n+1+1)l from
;  n  ih1:ex k fib n k
;  ih2:ex l fib(n+1)l

==> (drop 'ih1)

;goal symbol ?-side-premise-step-kernel-right-splithyp-dropped-side
;abbreviated by ?01
;ok, we now have the new goal
;?01-dropped: all k.fib n k -> ex l fib(n+1+1)l from
;  n  ih2:ex l fib(n+1)l

==> (assume 'k 'fibk)

;ok, under these assumptions we have the new goal
;?01-dropped-kernel: ex l fib(n+1+1)l from
;  n  ih2:ex l fib(n+1)l
;  k  fibk:fib n k
```

We repeat this for the second part of the induction hypothesis `ih2`.

```
==> (ex-elim 'ih2)

;ok, ?01-dropped-kernel can be obtained from
;?01-dropped-kernel-side: all l.fib(n+1)l -> ex l fib(n+1+1)l from
;  n  ih2:ex l fib(n+1)l
;  k  fibk:fib n k

==> (drop 'ih2)

;ok, we now have the new goal
;?01-dropped-kernel-side-dropped:
;                          all l.fib(n+1)l -> ex l fib(n+1+1)l from
;  n  k  fibk:fib n k

==> (assume 'l 'fibl)
```

```
;ok, under these assumptions we have the new goal
;?01-dropped-kernel-side-dropped-kernel: ex l fib(n+1+1)l from
;  n  k  fibk:fib n k
;  l  fibl:fib(n+1)l
```

Now we have `l` and `k` available and are ready to prove the existential quantifier using `ex-intro` by claiming that "`l + k`" is the required number.

```
==> (ex-intro (pt "k + l"))

;ok, ?01-dropped-kernel-side-dropped-kernel can be obtained from
;?01-dropped-kernel-side-dropped-kernel-main: fib(n+1+1)(k+l) from
;  n  k  fibk:fib n k
;  l  fibl:fib(n+1)l
```

The proof culminates in the following use of the `fibn` global assumption:

```
==> (use-with 'fibn 'n 'k 'l '?fibkl)

;ok, ?01-dropped-kernel-side-dropped-kernel-main can be obtained from
;?fibkl: fib n k & fib(n+1)l from
;  n  k  fibk:fib n k
;  l  fibl:fib(n+1)l
```

The rest is simple. We `split` the goal and use the two assumptions.

```
==> (split)

;ok, we have the new goals
;?fibkl-right: fib(n+1)l from
;  n  k  fibk:fib n k
;  l  fibl:fib(n+1)l

;?fibkl-left: fib n k from
;  n  k  fibk:fib n k
;  l  fibl:fib(n+1)l

==> (use-with 'fibk)

;ok, ?fibkl-left is proved. The active goal now is
;?fibkl-right: fib(n+1)l from
;  n  k  fibk:fib n k
;  l  fibl:fib(n+1)l

==> (use-with 'fibl)

;ok, ?fibkl-right is proved. Proof finished.
```

Now, if you have the distinct feeling that you deserve some sort of reward for bearing with me and do all this formal proofing for something obvious like Fibonacci Numbers, you are completely right.

- 1. Reward: From the proof obtained formally using MINLOG, we can automatically extract a clever program to compute the Fibonacci Numbers, as explained in the next section.
- 2. Reward: I will explain the automatic proofing commands `prop` and `search` in the "Summary of Commands" section below, even if we did not use them here.

3.3.3 Extracting Proofs from Programs

If you translate the specification of Fibonacci Numbers directly into a program you get something like

```
(define (fib n)
  (if (= n 0)  1
  (if (= n 1)  1
      (+ (fib (- n 2)) (fib (- n 1))))))
```

The function has exponential runtime. Even `(f1 30)` takes too long for me to finish before I run out of patience (Ctrl-C).
From the proof of an $\forall n \exists k f(n,k)$, MINLOG can extract automatically a program that computes the k for any given n. This is achieved with three functions:
`np` for "normalize proof" transforms the proof, which MINLOG stores in the variable `pproof` (for "partial proof"), into normal form.
`et` for "extract term" extracts the program term from the normal proof.
`nt` for "normalize term" finally reduces the extracted program term to its normal form.
Together this produces:

```
==> (nt (et (np pproof)))
;ok, variable nat*nat: nat*nat added
;ok, variable nat=>nat: nat->nat added
-: (lambda (n^1) (car ((((nat-rec-at (quote (star nat nat)))
(cons (num 1) (num 1))) (lambda (n^2) (lambda (nat*nat^3)
(cons (cdr nat*nat^3) ((plus-nat (car nat*nat^3))
(cdr nat*nat^3)))))) n^1)))
```

printed nicely:

```
 (lambda (n^1)
  (car ((((nat-rec-at (quote (star nat nat)))
          (cons (num 1) (num 1)))
         (lambda (n^2)
           (lambda (nat*nat^3)
             (cons (cdr nat*nat^3)
                   ((plus-nat (car nat*nat^3)) (cdr nat*nat^3))
             )
```

```
          )
        )
        ) n^1)
  )
 )
```

This is a scheme function. Rewriting variable names (e.g. `nat*nat^3` is a variable of type `nat*nat`, that is a pair of natural numbers), by ignoring types, and replacing the internal representation of the recursor by an implementation according to its definition ($\mathbf{R}ts0 = t$, $\mathbf{R}ts(n+1) = sn\mathbf{R}tsn$), this becomes:

```
(define recursor
  (lambda (base) (lambda (step) (lambda (n)
    (if (= n 0)
        base
        (( step (- n 1)) (((recursor base) step ) (- n 1)))
    )
  )))
)

(define fib
 (lambda (n)
  (car
      (((recursor
           (cons 1 1))
           (lambda (dummy)
            (lambda (pair)
             (cons (cdr pair)
                   (+ (car pair) (cdr pair))
             )
            ))
           n)
  )
 )
)
```

The inner loop defines a recursive function using the recursor. It accepts natural numbers and returns pairs of natural numbers. This function is called with n and the desired result is the first element of the return value pair. This is linear! You can run it directly.

```
==> (fib 3000)
-: 664390460366960072280217847866028384244163512452783259405
7976554262121416121925739644981098299982039113222680280946513
2446349331994409434926019045342723749188530316994678473551320
6351010996193829731816225856873369397843735278975554894868417
2613173381434012917562245042160510102589717323599066277020375
6438786517530547101123748849140252686120104032647025145598956
```

```
675902135010566909783124959436469825558314289701354227151784 6
028657107806246751070565698228205428466603218138388962758197 5
328137149180900441221912485637512169481172872421366781457732 6
618521478357661859018967313354840178403197559969056510791709 8
59144173304364898001
```

3.3.4 Summary of Commands

- (`prop`) searches automatically for a proof using minimal propositional logic. If no proof can be found, it adds the global assumption of "ex falso quodlibet", thereby obtaining intuitionistic propositional logic, and finally it will also add the global assumption of "stability", sufficient to provide the full power of classical propositional logic. Example (`prop`)

- (`search`) is an automatic proof searcher. It can be given arguments to guide the search process. Arguments might be, global and local assumptions with a natural number as a bound on the number of uses. Example (`search 5 '(ih 2)`)

- (`split`) will split a conjunction as goal into two separate goals.

- (`use-with` *formula* `'left`) applies the given formula, which must be a conjunction, and extracts the left part of the conjunction. `left` might be one of many arguments of an `use-with` command. Example: (`use-with 'stronger 'n 'left`)

- (`use-with` *formula* `'right`) is symmetric to the previous command.

- (`split-hyp` *assumption*) works like `split` but on the named assumption and produces two assumptions.

- (`ex-intro` *term*) is used if the current goal is an existential formula. It reduced this formula to a new goal by striping off the quantifier and replacing the quantified variable by the given *term*.

- (`ex-elim` *assumption*) is used if the *assumption* is an existential formula. It puts the given formula into the goal with an $\forall$-quantifier implying the old goal.

- (`ind`) uses induction to proof an $\forall$ quantified goal. It creates two new goals, the induction base, and the induction step.

- (`cut` *formula*) introduces a new formula into the proof. It replaces the old goal with two new goal, the given *formula* and the implication from the given *formula* to the old goal.

- (`add-global-assumption` *name formula*) adds a *formula* with the given *name* as a global assumption. Example: (`add-global-assumption 'fib0 (pf "fib 0 1")`)

- (`pt` *string*) parses a term. Example (`ex-intro (pt "k + 1")`)

- `pproof` The variable `pproof` (partial proof) is used by MINLOG to store the partially completed proof. After the proof is completed it is actually a complete proof, but still stored in the variable `pproof`.

- (`np` *proof*) will normalize the given *proof*. Normal proofs are required for some of the other functions below, for example `dp`. Example: `(np pproof)`
- (`et` *proof*) extracts the program term form a *proof*. Example: `(et (np pproof))`.
- (`nt` *term*) normalizes *terms* (similar to `np`, but for terms). Example: `(nt (et (np pproof)))`.
- (`dp` *proof*) displays a *proof*. Example: `(dp (np pproof))`.

4 MINLOG and Proof Theory

May be, some motivating final words on MINLOG and Proof Theory are in order, to answer two questions: Why do we look at these strange proofs, and why do we use MINLOG (isn't it complicated already?).
Look at Number Theory, for a long time a discipline which mathematicians pursued for mostly esthetic reasons and has become an important applied field of mathematics only very recently. Number Theory studies Numbers, but of course not individual numbers. It is pretty boring to know that 5 is a prime number. Number Theory looks for theorems about the totality of all numbers, like: There are infinitely many prime numbers. This is a theorem about all numbers, namely that among all numbers there is no largest prime number.
Similarly, Proof Theory investigates proofs. Again, proof theory is not investigating individual proofs, but all possible proofs. For example that arithmetic is a consistent theory is an interesting result of proof theory. It says that among all the proofs possible in the theory of arithmetic there is not a single one that derives a contradiction.
Most ordinary people, however, never encounter Number Theory, when it comes to numbers, they use a calculator. And here is another analogy: What a calculator is to Number Theory, MINLOG is to Proof Theory.

References

[1] O. Forster. Lmu scheme . `ftp://ftp.mathematik.uni-muenchen.de/pub/forster/sw/lmuscheme`.

[2] H. Schwichtenberg. Minlog manual . `http://www.mathematik.uni-muenchen.de/~schwicht/minlog/doc/manual.ps.gz`.

[3] H. Schwichtenberg. Minlog sources . `ftp://ftp.mathematik.uni-muenchen.de/pub/schwichtenberg/minlog`.

[4] H. Schwichtenberg. Minlog web page . `http://www.mathematik.uni-muenchen.de/~logik/minlog.html`.