# Introduction to Minlog

Franziskus Wiesnet

Marie Sklodowska-Curie fellow of the Istituto Nazionale di Alta Matematica

University of Trento, University of Verona

**Abstract**

In this chapter we would like to get the handling of the proof assistent Minlog across to the reader. Minlog is based on the Theory of Computable Functionals and it is based on the second chapter of my master's thesis [4].
On the website of the Minlog system [2] there are instructions for the download and installation of Minlog. We will use the dev branch. How Minlog can be changed to the dev branch is also described on this page.
In the Minlog file there is the folder `doc`, which contains the tutorial `tutor.pdf` and the reference manual `ref.pdf`. Some examples in this chapter are taken form the tutorial. In the reference manual many more commands for Minlog can be found.
Thanks to Helmut Schwichtenberg for proofreading this chapter.

## 1 Fundamental Commands in Minlog

### 1.1 Declaration of Predicate Variables

To deal with variables of any kind it is useful to declare them first. In the next example we need propositional variables, which are nullary predicate variables, to prove a theorem. A predicate variable can be declared with the command `add-pvar-name`. This command has the form

```
(add-pvar-name NAME (make-arity TYPES)).
```

Instead of `NAME` a list of strings is expected. For `TYPES` a list of types of the arguments should be inserted. In our case we would like to declare nullary predicate variables, which we denote as `A`, `B` and `C`. The list of types are therefore empty and we enter

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

As the output of Minlog we get

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

For demonstrations we declare a predicate variable `P` with one argument of type $\mathbb{N}$ and one argument of type $\mathbb{N} \to \mathbb{N}$:

```
(add-pvar-name "P" (make-arity (py "nat") (py "nat=>nat")))
```

As one might expect, `nat` is the name of the algebras of natural numbers. This algebra is considered more deeply in later sections. The arrow between types is denoted by `=>` and the command `(py STRING)` indicates that `STRING` should be interpreted as a type.
We also use often the commands `(pt STRING)`, `(pv STRING)` and `(pf STRING)`, which analogously indicates that `STRING` shoudl be interpreted as term, variable or formula.

## 1.2 First Proof

After the implementaion of the propositional variables `A`, `B` and `C` we use them to prove the theorem $(A \to B \to C) \to ((C \to A) \to B) \to A \to C$. A formal derivation of this short formula can be done easily. Writing a proof in Minlog does not mean giving a derivation tree of the formula. This would be too complex. Minlog processes the entered commands internally to a derivation tree, which can be displayed, as we will see.
At the beginning of each proof we enter the formula which we want to prove. We do this with the command

```
(set-goal FORMULA).
```

Instead of `FORMULA` we insert the formula in quotes. Therefore we wirte

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

As one sees, `->` stands for the implication arrow $\to$. As output we get

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

Below the dashed line there is the current goal formula. The assumptions which are given for the proof are located above the dashed line. We call this domain context. In this state of the proof there are no assumptions in the context but, since the goal formula is an implication with three premises, we move these premises as assumptions into the context and we prove the conclusion from these assumptions. The command to do this is

```
(assume NAMES).
```

Instead of `NAMES` we insert a list of names for the assumptions. Our list consists of three names. For example we write

```
(assume "assumption1" "assumption2" "assumption3")
```

and the output of the program is

```
ok, we now have the new goal

  assumption1:A -> B -> C
  assumption2:(C -> A) -> B
  assumption3:A
-------------------------------------------------------------
?_2:C
```

The three assumptions are in the context. If we had writen two names, we would only have the first two premises as assumptions in the context, and if we had writen more then three names, we would get an error message.
We now have to prove `C` by using the assumptions above the line. Since `assumption1` has `C` as conclusion, we ought to use this assumption. The general command for this is

```
(use ASSUMPTION),
```

whereby for `ASSUMPTION` we put in the name of the assumption which we would like to use. This assumption must have the goal formula as conclusion, otherwise we would get an error message. Therefore we enter

```
(use "assumption1").
```

The assumption with name `assumption1` has two premises. So Minlog requires a proof for each of these premises and we get two new goal formulas:

```
ok, ?_2 can be obtained from

  assumption1:A -> B -> C
  assumption2:(C -> A) -> B
  assumption3:A
-------------------------------------------------------------
?_4:B



  assumption1:A -> B -> C
  assumption2:(C -> A) -> B
  assumption3:A
-------------------------------------------------------------
?_3:A
```

First we have to prove the lower formula, which is exactly `assumption3`. Therefore we write

```
(use "assumption3").
```

The computer informs us that the lower formal is proven and we now have to give a prove of the upper formula:

```
ok, ?_3 is proved.  The active goal now is

  assumption1:A -> B -> C
  assumption2:(C -> A) -> B
  assumption3:A
-------------------------------------------------------------
?_4:B
```

To prove this goal we use `assumption2` by writing `(use "assumption2")`. Then we have to prove $A \to B$, which we do by moving $A$ into the context with `(assume "assumption4")` and using `assumption3` with `(use "assumption3")`. This finishes the proof and the output is

```
ok, ?_6 is proved.  Proof finished.
```

### 1.3 Representation of Proofs

After proving a formula Minlog is able to display this proof in different ways. With the command

```
(display-proof)
```

we get a representation which is similar to the derivation tree. An expansion of this command is

```
(cdp),
```

which is an abbreviation for `(check-and-display-proof)`. In addition to `display-proof` the proof is checked for correctness. If the proof is correct, the output is the same. In our case we get

```
.....A -> B -> C by assumption assumption1295
.....A by assumption assumption3297
....B -> C by imp elim
.....(C -> A) -> B by assumption assumption2296
......A by assumption assumption3297
.....C -> A by imp intro assumption4301
....B by imp elim
...C by imp elim
..A -> C by imp intro assumption3297
.((C -> A) -> B) -> A -> C by imp intro assumption2296
(A -> B -> C) -> ((C -> A) -> B) -> A -> C
 by imp intro assumption1295
```

It ought to be considered as a derivation tree. The number of dots at the beginning of each line indicates in which level of the derivation tree the subsequent formula is. After the formula there is the name of the applied derivation rule. A representation as derivation term is also possible with the commands

```
(proof-to-expr)
```

and

```
(proof-to-expr-with-formulas).
```

Both commands provide a derivation term of the proof. With the last command also the type of each variable is displayed. In our case the output of `(proof-to-expr-with-formulas)` is

```
assumption1: A -> B -> C
assumption2: (C -> A) -> B
assumption3: A
assumption4: C

(lambda (assumption1)
  (lambda (assumption2)
    (lambda (assumption3)
      ((assumption1 assumption3)
        (assumption2 (lambda (assumption4) assumption3))))))
```

which can easily be understood as derivation term.

## 1.4 Saving Theorems

Usually when a theorem is proven, one would like to use this theorem later again, maybe one reduces it on a special case or uses it as a lemma to prove a larger theorem. With the command

```
(save NAME)
```

the currently proven statement will be saved with the name `NAME`. In the previous section we proved the formula

```
(A -> B -> C) -> ((C -> A) -> B) -> A -> C
```

and now we can save it with

```
(save "Theorem1")
```

in Minlog. The output is

```
ok, Theorem1 has been added as a new theorem.
ok, program constant cTheoremOne: (alpha148=>alpha149=>alpha147)=>
((alpha147=>alpha148)=>alpha149)=>alpha148=>alpha147
of t-degree 1 and arity 0 added
```

When a theorem is saved we can use it in subsequent proofs, for example with `(use "Theorem1")`.
With the command

```
(display-theorems NAME)
```

Minlog shows us the theorem with name `NAME`. One can also insert a list of names instead of `NAME`, then all theorems with these names are shown. If we enter for example `(display-theorems "Theorem1")`, Minlog returns

```
Theorem1 (A -> B -> C) -> ((C -> A) -> B) -> A -> C
```

The `pritty-print` command, which has the form

```
(pp NAME),
```

displays also the formula with name `NAME`. But here the output is without the name

```
(A -> B -> C) -> ((C -> A) -> B) -> A -> C
```

and it takes only one argument. In contrast to `display-theorems` with `pp` it is posible to display any saved formula in the system and also terms can be shown with it. Therefore we often use `pp` later on.

## 1.5 Display Settings

Before proving new theorems in Minlog we would like to mention some useful commands, which help to get a clear representation of the output. The first statement, which we announce here, is

```
(set! COMMENT-FLAG BOOL).
```

If one inserts `#f` instead of `BOOL`, Minlog does not show comments anymore. Only irregularities are shown such as error messages and some warnings. Eliminating the comments is useful to enter many commands on a row, because the computer does not have to give an output and is therefore faster. If we place `#t` instead of `BOOL`, the comments are shown again.
To remove some assumptions from the context one can use the command

```
(drop STRINGS).
```

With this command Minlog does not display anymore the assumptions whose names are written instead of `STRINGS`. As an example in the last proof we had the output

```
ok, we now have the new goal

  assumption1:A -> B -> C
  assumption2:(C -> A) -> B
  assumption3:A
-------------------------------------------------------------
?_2:C
```

after promoting the premises in the context. If we enter

```
(drop "assumption1" "assumption2" "assumption3")
```

we get back

```
ok, we now have the new goal

-------------------------------------------------------------
?_3:C
```

but we can continue the proof analogously as above. Dropping assumptions can be helpful to make goals more readable, by removing useless assumptions. However, these are still present, they are just hidden.
With the instruction

```
(display STRING)
```

the text `STRING` is shown, even if `(set! COMMENT-FLAG #f)` was entered. In this way one can point out important declarations, definitions or theorems. In the example above it may be useful to enter

```
(display "A, B and C are now propositional variable")
```

to point out that `A`, `B` and `C` are now taken. With

```
(newline)
```

one can make a line break.
As a last resort the command

```
(undo)
```

is to mention. With it one can revert the last step in a proof. Then Minlog shows the state before the last command.

## 1.6 Loading External Files

The folder `lib` of the Minlog file contains very helpful data files. One of them is `nat.scm`. There the natural numbers are introduced, some functions on the natural numbers like $+$, $*$ and the Boolean ones $\leq$ and $<$ are defined and many simple properties of these are proven. For ones own proofs one would like to use this preparatory work. With entering

```
(libload NAME)
```

the file `NAME` from folder `lib` is loaded into the system. It is recommended to use `(set! COMMENT-FLAG #f)` before loading some files. For loading `nat.scm` we write

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

A general version of `libload` is the command `load`. For example

```
(load "lib/nat.scm")
```

is equivalent to `(libload "nat.scm")`.

## 1.7 Proofs in Predicate Logic

Now we would like to prove formulas with universal quantifiers. As first example we take the formula $\forall_n(Pn \to Qn) \to \forall_n Pn \to \forall_n Qn$ where $n$ has type $\mathbb{N}$ and $P$, $Q$ are unary predicate variables. First we load `nat.scm` as in the section above. In this file `nat` is set as the name of the algebra of natural numbers and `n`, `m` are already declared as variables of type `nat`. We define `P` and `Q` via the command

```
(add-pvar-name "P" "Q" (make-arity (py "nat"))).
```

and set goal formula with

```
(set-goal "all n(P n -> Q n) -> all n P n -> all n Q n").
```

As one can see, the universal quantifier is denoted by `all` and requests two arguments. The first argument is the variable to be quantified in the formula given by the second argument. Minlog returns

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

With

```
(assume "assumption1" "assumption2")
```

we move the premises into the context. Then the goal formula is `all n Q n`. To prove it we put a new variable into the context and prove the formula specialized to this variable. Therefore we enter

```
(assume "m")
```

and the output is

```
 ok, we now have the new goal

  assumption1:all n(P n -> Q n)
  assumption2:all n P n
  m
-------------------------------------------------------------
?_3:Q m
```

Of course, we could also write `(assume "n")` to get the goal formula `Q n`. But it has always to be a variable of type `nat`. Therefore `(assume "a")` would lead to an error message.
To prove `Q m` we use `assumption1` via `(use "assumption1")`, since it is an abstraction of `Q m`. As output we get

```
 ok, ?_3 can be obtained from

  assumption1:all n(P n -> Q n)
  assumption2:all n P n
  m
-----------------------------------------------------------------
?_4:P m
```

and with (use "assumption2") the proof is already finished.

## 1.8 use-with

Sometimes Minlog does not recognize how a formula in the use-command has to be specialized. In such cases one has to tell directly how to use a formula. To do this there is the instruction

```
(use-with NAME LIST).
```

For NAME one inserts the name of the formula which should be specialised. Instead of LIST we need a list of formulas and terms. For each universal quantifier the corresponding term TERM have to be stated with (pt TERM). For each premise one has to provide the name of an assumption of this formula. Optionally one can write "?" for a premise. Then Minlog requires a proof of this premise afterwards.
As an example we consider the proof of the last section. After the instructions

```
(set-goal "all n(P n -> Q n) -> all n P n -> all n Q n")
(assume "assumption1" "assumption2")
(assume "m")
```

we have the output

```
ok, we now have the new goal

  assumption1:all n(P n -> Q n)
  assumption2:all n P n
  m
-----------------------------------------------------------------
?_3:Q m
```

Here the goal formula is a specialization of assumption1. So instead of (use "assumption1") we can also write

```
(use-with "assumption1" (pt "m") "?")
```

and the output again is

```
 ok, ?_3 can be obtained from

  assumption1:all n(P n -> Q n)
```

```
  assumption2:all n P n
  m
-----------------------------------------------------------------
?_4:P m
```

Note that for each universal quantification and each implication in front of the goal formula there has to be exactly one argument in the `use-with` command. The goal formula `P m` can be proven with `(use "assumption2")` or with `(use-with "assumption2" (pt "m"))`. Here one sees that the `use` instruction is a short form of `use-with` and actually one only needs `use-with`, if Minlog does not recognize how to use `use`.

## 1.9 inst-with

The instruction `inst-with` is similar to the command `use-with`. One also specializes a formula. But the specialized formula does not have to be the goal formula but it is added to the context. The command has the form

```
(inst-with NAME LIST).
```

Analogously to `use-with`, `NAME` has to be the name of the formula which will be specialized and `LIST` is a list of formula names and terms. As example we take the initial situation of the last section:

```
 ok, we now have the new goal

  assumption1:all n(P n -> Q n)
  assumption2:all n P n
  m
-----------------------------------------------------------------
?_3:Q m
```

Now we specialize `assumption2` to `m` with the command

```
(inst-with "assumption2" (pt "m"))
```

and get

```
ok, ?_3 can be obtained from

  assumption1:all n(P n -> Q n)
  assumption2:all n P n
  m  3:P m
-----------------------------------------------------------------
?_4:Q m
```

As we see, the specialized formula appears in the context and has the name `3`. If we use this formula later on, we must not put `3` in quotes.
If one would like to give a name to the new formula one can use the command

```
(inst-with-to NAME LIST NAME1).
```

The arguments NAME and LIST are analog to the corresponding ones in `inst-with` but NAME1 is the name of the new formula in the context. In our example we can write

```
(inst-with-to "assumption1" (pt "m") 3 "goal")
```

to get the output

```
ok, ?_4 can be obtained from

  assumption1:all n(P n -> Q n)
  assumption2:all n P n
  m  3:P m
  goal:Q m
-----------------------------------------------------------------
?_6:Q m
```

and by entering `(use "goal")` the proof is finished.

## 1.10 assert and cut

Many proofs, especially long proofs, one would like to split up into different parts such that in each part another formula is proven. In informal proofs this is done by expressions like “Claim:...” or “It is sufficient to prove ...”. To do this in Minlog there are the two commands

```
(assert FORMULA)
```

and

```
(cut FORMULA).
```

The proof of the goal formula GOAL is devided by both commends into a proof of FORMULA and a proof of FORMULA->GOAL. The difference between these two commands are the order of the new parts. By using `assert` Minlog firstly requires a proof of FORMULA and afterwards a proof of FORMULA->GOAL. By using `cut` it is reversed.
In the example of the previous sections after the output

```
ok, we now have the new goal

  assumption1:all n(P n -> Q n)
  assumption2:all n P n
  m
-----------------------------------------------------------------
?_3:Q m
```

we can enter `(assert "P m")` and get

```
 ok, ?_3 can be obtained from

  assumption1:all n(P n -> Q n)
  assumption2:all n P n
  m
-------------------------------------------------------------
?_4:P m -> Q m

  assumption1:all n(P n -> Q n)
  assumption2:all n P n
  m
-------------------------------------------------------------
?_5:P m
```

back. The new goal can be proven with `(use "assumption2")` followed by `(use "assumption1")`. The output after using `(cut "P m")` is

```
ok, ?_3 can be obtained from

  assumption1:all n(P n -> Q n)
  assumption2:all n P n
  m
-------------------------------------------------------------
?_5:P m



  assumption1:all n(P n -> Q n)
  assumption2:all n P n
  m
-------------------------------------------------------------
?_4:P m -> Q m
```

These are the same goals in reversed order.

## 1.11 Proof Search

Minlog is able to search for proofs of formulas by itself. Of course, this is only successful for short proofs. The command to search for the current goal is

```
(search).
```

In the last sections we have seen many unnecessarily long proofs of `all n(P n -> Q n) -> all n P n -> all n Q n`. Now we see the shortest proof: after entering

```
(set-goal "all n(P n -> Q n) -> all n P n -> all n Q n")
(search)
```

we get the output

```
ok, ?_1 is proved by minimal quantifier logic.  Proof finished.
```

and the proof is done. The probability that a proof is found by `search` is indirect proportional to the length of the proof. Especially the probability is very low if Minlog has to find a witness for an existential quantifier or if it has to use not only the context formulas but some other theorems.
To use the `search` command several times in a row one can use the instruction

```
(auto).
```

This command enters the command `(search)` until the proof is done or `(search)` does not find a proof anymore.

## 1.12 Cheating in Minlog

It often occurs that it takes very long to prove formulas by Minlog although they are obviously true for humans. In such cases it could be reasonable to declare these formulas as globals assumptions first and prove them later. By the instruction

```
(add-global-assumption NAME FORMULA)
```

the formula `FORMULA` is saved under the name `NAME`. After adding this formula as a global assumption it can be used in each proof, for example by the `use` command. The instruction

```
(display-global-assumption NAME)
```

shows the global assumption with name `NAME`. Instead of `NAME` one can also put in a list of names. Then all global assumptions in this list are shown. If this list is empty, all global assumptions are shown. If we enter `(display-global-assumption)` directly after starting Minlog, we get the output

```
Stab ((Pvar -> F) -> F) -> Pvar
Efq F -> Pvar
StabLog ((Pvar -> bot) -> bot) -> Pvar
EfqLog bot -> Pvar
```

We see, that the ex-falso-quodlibet and the stability are global assumptions by default.
The command

```
(remove-global-assumption LIST)
```

removes the global assumptions in list `LIST`. Therefore, if we want to use Minlog without any global assumptions, we accomplish this with the command

```
(remove-global-assumption "Stab" "Efq" "StabLog" "EfqLog").
```

While proving a formula it sometimes occurs that one would like to set a subgoal as a global assumption. In such cases one can use the command

```
(admit).
```

By entering `(admit)` the current goal formula becomes a global assumption and is considered to be proven. In this way one can actually prove each formula, which justifies the title of this section. Therefore it is recommended that a finished proof contains `admit` or a global assumption as rarely as possible.

### 1.13 Searching in Minlog

While doing a proof about a certain issue, for example about the addition on naturals numbers, one often wants to know which statements about this issue are already given. A list of the given statements can be displayed by the command

```
(search-about LIST).
```

Instead of `LIST` Minlog requires a list of strings and the output are all theorems and global assumptions whose names contain each of the strings in the list. Therefore the name of a theorem should always be as meaningful as possible.
Now if `nat.scm` is loaded, we can search for theorems about the addition on naturals numbers by the command

```
(search-about "Nat" "Plus").
```

As ouput we get

```
Theorems with name containing Nat and Plus
NatPlusDouble
all n,m NatDouble n+NatDouble m=NatDouble(n+m)
.
.
.
NatPlusComm
all n,m n+m=m+n
No global assumptions with name containing Nat and Plus
```

which is a long list of theorems and an empty list of global assumptions. Of course, instead of the dots there are many more theorems.

## 2 Algebras and Inducively Defined Predicates

In this section we introduce algebras and inductively defined predicates in Minlog.

## 2.1 Algebras

The general command to define algebras in Minlog is given by

```
(add-algs NAME LISTS).
```

Here `NAME` is the name of the defined algebra in quotes and `LISTS` is a List of pais

```
'(NAME TYPE)
```

with a name for the constructor of type `TYPE`. Here every constructor gets his name during the definition of the algebra. We will consider some examples: in `nat.scm` the algebra of natural numbers is given by

```
(add-algs "nat" '("Zero" "nat") '("Succ" "nat=>nat")).
```

As we see, zero is denoted by `Zero` and the successor is denoted by `Succ`. By entering the instruction

```
(display-alg NAME)
```

Minlog shows us a list of the constructors of the algebra `NAME`. In our case `(display-alg "nat")` leads to the output

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

The boolean algebra is implemented in Minlog by default. Its two constructors have the names `True` and `False`. We can see this see by entering `(display-alg "boole")`:

```
boole
True: boole
False: boole
```

In Minlog we can also introduce algebras with parameters. The strings `alpha`, `beta`, `gamma`, `alpha0`, `beta0`, `gamma0` and so on stand for type variables. In the library file `list.scm` the algebra of lists is defined by

```
(add-algs "list" '("list" "Nil") '("alpha=>list=>list" "Cons")).
```

The list type has the paramenter `alpha`. By using the list algebra this parameter has to be written explicitly. We see this by displaying the algebra with `(display-alg "list")`:

```
list
Nil: list alpha
Cons: alpha=>list alpha=>list alpha
```

## 2.2 Declaration of Term Variables

Before using variables Minlog has to know which variables have which type. For each type with name TYPE the string TYPE is by default also the name of a variable with the same type. Therefore a command like

```
(set-goal "all nat nat=nat")
```

can be understood by Minlog.
If v is a variable of type TYPE, then also v0, v1 and so on are variables of type TYPE. Hence

```
(set-goal "all nat1000 nat1000=nat1000")
```

is also a valid instruction.
Of course, it is also possible to add new names for variables. The command to do this is

```
(add-var-name NAMES TYPE).
```

For NAMES we give a list of the variables names which should have type TYPE. In nat.scm the letters n, m and l are introduced as variables names for natural numbers by the instruction

```
(add-var-name "n" "m" "l" (py "nat")).
```

## 2.3 Inductively Defined Predicates

Inductively defined predicates are defined by the command

```
(add-ids (list (list NAME (make-arity TYPES) ALGNAME)) LIST).
```

NAME stands for the name of the defined predicate. The types of the arguments of the predicate are inserted for TYPES. ALGNAME is the name of the algebra of the inductively defined predicate and the introduction rules of the predicate are written instead of LIST in the form

```
'(FORMULA NAME),
```

whereby FORMULA is the introduction axiom and NAME its name.
As an example, we define the unary inductively defined predicate EvenI, which says that a natural number is even. To define the property that a natural numer is even, we define that 0 is even and, if n is even, then also n+2 is even. In Minlog we do this as follows:

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

With the command (display-alg "algEvenI") we take a look at the algebra of EvenI:

```
algEvenI
CInitEvenI: algEvenI
CGenEvenI: nat=>algEvenI=>algEvenI
```

We see that the constructor which corresponds to the introduction axiom `Axiom` has the name `CAxiom`. This holds in general.
One also sees that `algEvenI` and `nat` have the same constructor types. Therefore one could also write `nat` instead of `algEvenI` in the definition of `EvenI`. In general if one writes an already defined algebra instead of `ALGNAME`, Minlog checks whether the constructor types are the same. If this is the case this algebra is used as the type of the predicate.
By the command

```
(display-idpc NAME)
```

the introduction axioms of the inductively defined predicate `NAME` and its algebra are shown.
For the predicate `EvenI` we enter `(display-idpc "EvenI")` and get

```
EvenI with content of type algEvenI
InitEvenI: EvenI 0
GenEvenI: all n(EvenI n -> EvenI(Succ(Succ n)))
```

back.
The introduction axioms are saved as theorems in the system und can be used for example with the `use` command. There is also the possibility to use the instruction

```
(intro N).
```

Then Minlog uses the `N`-th introduction axiom of the predicate in the goal formula. The numbering starts with 0.

## 2.4 Proofs with Inductively Defined Predicates

As a counterpart to `EvenI` we define the predicate `OddI` with says that a natural number is odd:

```
(add-ids
 (list (list "OddI" (make-arity (py "nat")) "algOddI"))
 '("OddI 1" "InitOddI")
 '("all n(OddI n -> OddI(Succ( Succ n)))" "GenOddI"))
```

We prove the theorem that the successor of an even number is odd. Therefore we enter the instruction

```
(set-goal "all n(EvenI n -> OddI(Succ n))").
```

By using `(assume "n" "EvenIn")` we put the variable `n` and the premise `EvenI n` into the context and the output is

```
ok, we now have the new goal

  n  EvenIn:EvenI n
-----------------------------------------------------------------
?_2:OddI(Succ n)
```

To prove the goal formula we would like to apply the elimination axiom of `EvenI` to the predicate $\{n|\texttt{OddI}(\texttt{Succ}\ n)\}$. In general, if `NAME` is the name of a formula which is an inductively defined predicate, we can enter the command

```
(elim NAME)
```

to use the elimination axiom of this predicate. The goal formula becomes the parameter in the elimination axiom and exactly the free variables in the formula `NAME` are abstracted. The elimination axiom of `EvenI` with parameter $P$ is given by

$$\forall_n(\texttt{EvenI}\ n \to Pn \to \forall_n(\texttt{EvenI}\ n \to Pn \to P(SSn)) \to Pn).$$

Since `EvenI` $n$ is already given, Minlog requires a proof of $P0$ and $\forall_n(\texttt{EvenI}\ n \to Pn \to P(SSn))$. Here $Pn := \texttt{OddI}\ Sn$ and indeed after `(elim "EvenIn")` we get the output:

```
ok, ?_2 can be obtained from

  n  EvenIn:EvenI n
-----------------------------------------------------------------
?_4:all n(EvenI n -> OddI(Succ n) -> OddI(Succ(Succ(Succ n))))



  n  EvenIn:EvenI n
-----------------------------------------------------------------
?_3:OddI 1
```

For the goal `OddI 1` we just write `(use "InitOddI")`. To prove the other goal we move with

```
(assume "m" "EvenIm" "OddISm")
```

the variable and the two premises into the context. The new goal is given by:

```
ok, we now have the new goal

  n  EvenIn:EvenI n
  m  EvenIm:EvenI m
  OddISm:OddI(Succ m)
-----------------------------------------------------------------
?_5:OddI(Succ(Succ(Succ m)))
```

We prove it by the second introduction axiom of `OddI`, which is given by

```
GenOddI: all n(OddI n -> OddI(Succ(Succ n))).
```

So we enter `(use "GenOddI")` followed by `(use "OddISm")` and the proof is finished. With `(cdp)` we take a look at the proof tree:

```
 ......allnc n8136(EvenI n8136 -> OddI 1 ->
        all n(EvenI n -> OddI(Succ n) -> OddI(Succ(Succ(Succ n))))
        -> OddI(Succ n8136)) by axiom Elim
......n
.....EvenI n -> OddI 1 -> all n(EvenI n -> OddI(Succ n) ->
     OddI(Succ(Succ(Succ n)))) -> OddI(Succ n) by allnc elim
.....EvenI n by assumption EvenIn3363
....OddI 1 -> all n(EvenI n -> OddI(Succ n) ->
    OddI(Succ(Succ(Succ n)))) -> OddI(Succ n) by imp elim
....OddI 1 by axiom Intro
...all n(EvenI n -> OddI(Succ n) -> OddI(Succ(Succ(Succ n)))) ->
   OddI(Succ n) by imp elim
........all n(OddI n -> OddI(Succ(Succ n))) by axiom Intro
........Succ m
.......OddI(Succ m) -> OddI(Succ(Succ(Succ m))) by all elim
.......OddI(Succ m) by assumption OddISm3367
......OddI(Succ(Succ(Succ m))) by imp elim
.....OddI(Succ m) -> OddI(Succ(Succ(Succ m)))
     by imp intro OddISm3367
....EvenI m -> OddI(Succ m) -> OddI(Succ(Succ(Succ m)))
    by imp intro EvenIm3366
...all n(EvenI n -> OddI(Succ n) -> OddI(Succ(Succ(Succ n))))
   by all intro
..OddI(Succ n) by imp elim
.EvenI n -> OddI(Succ n) by imp intro EvenIn3363
all n(EvenI n -> OddI(Succ n)) by all intro
```

Although the output is complex, we see that the first formula is derived by the elimination axiom and the formulas `all n(OddI n -> OddI(Succ(Succ n)))` and `OddI 1` are derived by the introduction axioms. This goes well, because we have used the elimination axiom exactly once and the introduction axioms exactly twice.

# 3 Decorations

Minlog allows usage of decorated connectives $\to^{nc}$ and $\forall^{nc}$ to indicate that the premise or the quantified variable are not used computationally. Their introduction and elimination rules are are like the ones for $\to$ and $\forall$, except that in the introduction rules $\to^{nc+}$ and $\forall^{nc+}$ the premise or the quantified variable are not used computationally. Since these are rather intuitive concepts, we do

not develop the theory here but only explain their usage in Minlog.
Especially when dealing with decorated logical symbols computer support is very useful. As just explained, we have to calculate the free computational variables, if we want to use the rules $\to^{nc+}$ and $\forall^{nc+}$. With computer support this is trivial but to do it with pen and paper takes some time.

### 3.1 Non-computational Universal Quantifier

We first take a look at the non-computational universal quantifier. In Minlog it is denoted by `allnc` and has the same syntactical rules as the normal universal quantifier. In section 1.7 we have proven the formula $\forall_n(Pn \to Qn) \to \forall_n Pn \to \forall_n Qn$ and now we would like to prove the decorated form $\forall_n^{nc}(Pn \to Qn) \to \forall_n^{nc} Pn \to \forall_n^{nc} Qn$. In order to do this we load the library file `nat.scm` and declare the predicates `P` and `Q` as we have done in section 1.7. Then we enter the instruction

```
(set-goal "allnc n(P n -> Q n) -> allnc n P n -> allnc n Q n").
```

The output is as expected:

```
-------------------------------------------------------------
?_1:allnc n(P n -> Q n) -> allnc n P n -> allnc n Q n
```

By the command `(assume "assumption1" "assumption2" "m")` we move the two premises and the variable `m` into the context. The output is similar to the one we have had in section 1.7:

```
ok, we now have the new goal

  assumption1:allnc n(P n -> Q n)
  assumption2:allnc n P n
  {m}
-------------------------------------------------------------
?_2:Q m
```

The difference is that the variable `m` is in curly brackets. This tells us that `m` should not be used computationally. In our case both quantifiers in the assumptions are non-computational. Therefore we can enter `(use "assumption1")` followed by `(use "assumption2")` without any problems and the proof is done. The output after `(cdp)` is

```
.....allnc n(P n -> Q n) by assumption assumption12189
.....m
....P m -> Q m by allnc elim
.....allnc n P n by assumption assumption22190
.....m
....P m by allnc elim
...Q m by imp elim
..allnc n Q n by allnc intro
```

```
.allnc n P n -> allnc n Q n by imp intro assumption22190
allnc n(P n -> Q n) -> allnc n P n -> allnc n Q n
by imp intro assumption12189
```

We see that the rules `allnc elim` and `allnc intro` occur. These are the rules for the non-computational universal quantifier.
While dealing with the non-computational universal quantifier one should note that Minlog just returns a warning if one applies a non-computational variable to a computational universal quantifier. The proof can be continued normally. As an example we try to prove the formula above where the first universal quantifier is changed to a non-computational one:

```
(set-goal "all n(P n -> Q n) -> allnc n P n -> allnc n Q n")
```

Here we also start with `(assume "assumption1" "assumption2" "m")`. If we now enter `(use "assumption1")`, we just get a warning:

```
Warning: nc-intro with cvar(s)
m
ok, ?_2 can be obtained from

  assumption1:all n(P n -> Q n)
  assumption2:allnc n P n
  {m}
-------------------------------------------------------------
?_3:P m
```

Minlog does not stop the proof with an error message, because at this point it is not clear whether usage of the rule was really incorect. If we finish the "proof" with `(use "assumption2")` and enter `(cdp)`, we will be informed that the "proof" is incorect:

```
warning: allnc-intro with cvarm
.....all n(P n -> Q n) by assumption assumption12193
.....m
....P m -> Q m by all elim
.....allnc n P n by assumption assumption22194
.....m
....P m by allnc elim
...Q m by imp elim
..allnc n Q n by allnc intro
.allnc n P n -> allnc n Q n by imp intro assumption22194
all n(P n -> Q n) -> allnc n P n -> allnc n Q n
by imp intro assumption12193
Incorrect proof: nc-intro with computational variable(s)
m
```

Here we also see a difference between `(cdp)` and `(dp)`. The command `(dp)` would not inform the user that the proof is incorrect.

## 3.2 Non-computational Implication

The non-computational implication is denoted by `-->` and has the same syntactical rules as `->`. As an application example we show that the non-computational implication is transitive. Therefore we introduce three propositional variables by `(add-pvar-name "A" "B" "C" (make-arity))` and set the goal formula:

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

As first step we move the three assumptions into the context with the command `(assume "assumption1" "assumption2" "assumption3")`. The output of Minlog is

```
ok, we now have the new goal

  {assumption1}:A --> B
  assumption2:B --> C
  {assumption3}:A
--------------------------------------------------------------
?_2:C
```

Similar to the non-computational variables we see that the non-computational assumptions are in curly brackets. These assumptions can be only used in non-computational parts of the proof. Whether we are in a non-computational or not is not shown by Minlog and must be checked by the user.
In our example we first use `assumption2`, which is not in curly brackets, with `(use "assumption2")`. According to `assumption2` `B` implies `C` non-computationally. Therefore while proving `B` we are in a non-computational part of the proof, hence we can enter `(use "assumption1")` followed by `(use "assumption3")` which finishes the proof. We see that the proof is correct, if we enter `(cdp)`:

```
....B --> C by assumption assumption22198
.....A --> B by assumption assumption12197
.....A by assumption assumption32199
....B by impnc elim
...C by impnc elim
..A --> C by impnc intro assumption32199
.(B --> C) -> A --> C by imp intro assumption22198
(A --> B) --> (B --> C) -> A --> C by impnc intro assumption12197
```

The proof tree uses the new rules `impnc intro` and `impnc elim`.
It should be noted that one implication in the proven formula is computational otherwise we could not prove the formula, because `C` could have computational content and if every implication were non-computational, this content would come from nothing.
Analogously to the non-computational universal quantifier Minlog only returns a warning if one uses a non-computational assumption in a computational part.

## 3.3 Decorated Predicates

The non-computational universal quantifier and implication can be used for the definition on an inductively defined predicate. In section 2.3 we have defined the predicate `EvenI` with the second introduction axiom

```
GenEvenI: all n(EvenI n -> EvenI(Succ (Succ n)))
```

This axiom can be changed to

```
GenEvenI: allnc n(EvenI n -> EvenI(Succ (Succ n)))
```

since heuristically one can say that the information on `n` is already given in `EvenI n`. Compared to inductively defined predicate without decoration there is not an essential change.
Also the definition of a non-computational inductively defined predicate is quite similar to the definition of a computational one. The only change is that one does not declare a name for the algebra of the predicate.
To give a good example we use the algebra of lists, which are introduced in the library file `list.scm`. We would like to define the predicate `RevI` which takes two lists as arguments and says that the first list reversed is the second list. Therefore we load `nat.scm` and `list.scm` form the library. The list type is defined with a type variable `alpha` as we have seen in section 2.1. We first declare two variables `xs` and `ys` of type `list alpha` and one variable `x` of type `alpha`. In `list.scm` also the infix notation `::` for the constructor `Cons alpha` is declared. For `x ::(Nil alpha)` we have the abbreviation `x:` and `++` denotes the concatenation of two lists. The predicate `revI` is therefore defined by the command:

```
(add-ids (list (list "RevI" (make-arity (py "list alpha")
                                        (py "list alpha"))))
 '("RevI(Nil alpha)(Nil alpha)" "InitRevI")
 '("all xs,ys,x(RevI xs ys -> RevI(xs++x:)(x::ys))" "GenRevI"))
```

As an application we prove that `RevI` is symmetric. Hence we enter

```
(set-goal "all xs,ys(RevI xs ys -> RevI ys xs)").
```

After the command `(assume "xs" "ys" "RevIxsys")` the output is

```
ok, we now have the new goal

  xs  ys  RevIxsys:RevI xs ys
---------------------------------------------------------------
?_2:RevI ys xs
```

Here we use the elimination rule for `RevI xs ys` to the goal formula by the instrution `(elim "RevIxsys")`. This is possible since `RevI` is non-computational. Minlog gives

```
ok, ?_2 can be obtained from

  xs  ys  RevIxsys:RevI xs ys
-------------------------------------------------------------
?_4:all xs,ys,x(RevI xs ys -> RevI ys xs
                -> RevI(x::ys)(xs++x:))



  xs  ys  RevIxsys:RevI xs ys
-------------------------------------------------------------
?_3:RevI(Nil alpha)(Nil alpha)
```

back. The goal formula is easily proven by `(use "InitRevI")`. To prove the second formula we move via

```
(assume "xs1" "ys1" "x" "RevIxs1ys1" "RevIys1xs1")
```

everything into the context and again use the elimination axiom for the assumption `RevIys1xs1` with `(elim "RevIys1xs1")`. The output is

```
ok, ?_5 can be obtained from

  xs  ys  RevIxsys:RevI xs ys
  xs1  ys1  x  RevIxs1ys1:
    RevI xs1 ys1
  RevIys1xs1:RevI ys1 xs1
-------------------------------------------------------------
?_7:all xs,ys,x0(
     RevI xs ys -> RevI(x::xs)(ys++x:)
     -> RevI(x::xs++x0:)((x0::ys)++x:))


  xs  ys  RevIxsys:RevI xs ys
  xs1  ys1  x  RevIxs1ys1:
    RevI xs1 ys1
  RevIys1xs1:RevI ys1 xs1
-------------------------------------------------------------
?_6:RevI(x:)((Nil alpha)++x:)
```

We know that `:x` and `(Nil alpha)++x:` are equal terms and Minlog also knows it. Form `InitRevI` we get `RevI(Nil alpha)(Nil alpha)` and with GenRevI this leads to `RevI :x :x`. Since Minlog does not recognize how to apply the `use` command we enter

```
(use-with "GenRevI" (pt "(Nil alpha)") (pt "(Nil alpha)")
                                        (pt "x") "InitRevI").
```

After this we get

```
ok, ?_6 is proved.  The active goal now is

  xs  ys  RevIxsys:RevI xs ys
  xs1  ys1  x  RevIxs1ys1:
    RevI xs1 ys1
  RevIys1xs1:RevI ys1 xs1
-------------------------------------------------------------
?_7:all xs,ys,x0(
     RevI xs ys -> RevI(x::xs)(ys++x:)
     -> RevI(x::xs++x0:)((x0::ys)++x:))
```

back. `RevI(x::xs)(ys++x:)->RevI(x::xs++x0:)((x0::ys)++x:))` has the form of the axiom `GenRevI`. Therefore we move with `(assume "xs2" "ys2" "x0" "assumption1")` everything but the last premise into the context and finish the proof by

```
(use-with "GenRevI" (pt "x::xs2") (pt "ys2++x:") (pt "x0")).
```

Finally we take a look at the predicate `RevI` by `(display-idpc "RevI")` and we see that `RevI` is non-computational:

```
RevI non-computational
InitRevI: RevI(Nil alpha)(Nil alpha)
GenRevI: all xs,ys,x(RevI xs ys -> RevI(xs++x:)(x::ys))
```

## 3.4 Leibniz Equality and Simplification

An important example of a non-computational inductively defined predicate is the Leibniz equality. It is saved in Minlog by default and denoted by `EqD`. We see its introduction axiom if we enter `(display-idpc "EqD")`:

```
EqD non-computational
InitEqD: allnc alpha^ alpha^ eqd alpha^
```

The system also uses the infix notation `T1 eqd T2` instead of `EqD T1 T2`. Since the type of the computational version of the Leibniz equality would be the unit type each predicate, not only the non-computational ones, can be used in the elimination axiom.
The characteristic property of the equality is $\forall_{x,y}(\mathsf{Eq}xy \to A(x) \to A(y))$. This perperty can be used conveniently in Minlog by the `simp` command. If we have a formula `FORMULA` of the form `T1 eqd T2`, which is a Leibniz equality or an abstraction of it, with the command

```
(simp FORMULA)
```

each term `T1` in the current goal formula is replaced by `T2`. If `FORMULA` has premises, Minlog requires also a proof of these premises. Formally the theorem $\forall_{x,y}(\mathsf{Eq}xy \to A(y) \to A(x))$, which is obviously equivalent to the characteristic property of the equality above, is used. The first premise $\mathsf{Eq}xy$ is already given

and Minlog requires a proof of $A(y)$.
As an example we prove for a natural number $n$ and a function $f : \mathbb{N} \to \mathbb{N}$ that $f(n) = n \to f(n) = f(f(n))$ holds. Hence we enter

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

With `(assume "f" "n" "fn=n")` we move the variables and the premise into the context and the new goal is:

```
ok, we now have the new goal

  f  n  fn=n:f n eqd n
-----------------------------------------------------------------
?_2:f(f n)eqd f n
```

Here we directly use `(simp "fn=n")`, which leads to the output

```
ok, ?_2 can be obtained from

  f  n  fn=n:f n eqd n
-----------------------------------------------------------------
?_3:f n eqd n
```

and the proof is finished by `(use "fn=n")`.
It is also possible to use the other direction of the equality, i.e., the theorem $\forall_{x,y}(\mathsf{Eq}xy \to A(x) \to A(y))$. One can do this by the command

```
(simp "<-" FORMULA).
```

In our proof above we could also use this command, even though it is not expedient. Nevertheless, for demonstration purposes we show the output after entering `(simp "<-" "fn=n")` instead of `(simp "fn=n")`:

```
ok, ?_2 can be obtained from

  f  n  fn=n:f n eqd n
-----------------------------------------------------------------
?_3:f(f(f n))eqd f(f n)
```

Each occurrence of `n` was replaced by `f n`.
It is not only possible to insert the name of an already known formula for `FORMULA`. One can also write `(pf EQUALITY)` instead of `FORMULA`, where `EQUALITY` has to be an abstraction of `T1 eqd T2`. In this case Minlog also requires a proof of `EQUALITY`.

## 3.5 Examples of Inductively Defined Predicates

In this section we discuss disjunction, conjunction and the existential quantifier in Minlog. We start with conjunction. In Minlog there are four versions of it:

```
AndD with content of type yprod
InitAndD: Pvar1 -> Pvar2 -> Pvar1 andd Pvar2
AndL with content of type identity
InitAndL: Pvar1 -> Pvar2 --> Pvar1 andl Pvar2
AndR with content of type identity
InitAndR: Pvar1 --> Pvar2 -> Pvar1 andr Pvar2
AndNc non-computational
InitAndNc: Pvar1 --> Pvar2 --> Pvar1 andnc Pvar2
```

There is no `AndU` since the type of it would be the unit type and therefore `AndU` and `AndNc` are equivalent. To use the introduction axiom of each conjuction one also has the command `(split)`. For the predicates above `(split)` and `(intro 0)` are synonyms.
Similar to conjunction the existential quantifier also has four versions:

```
ExD with content of type yprod
InitExD: all alpha^((Pvar alpha)alpha^ ->
                  exd alpha^0 (Pvar alpha)alpha^0)
ExL with content of type identity
InitExL: all alpha^((Pvar alpha)alpha^ -->
                  exl alpha^0 (Pvar alpha)alpha^0)
ExR with content of type identity
InitExR: allnc alpha^((Pvar alpha)alpha^ ->
              exr alpha^0 (Pvar alpha)alpha^0)
ExNc non-computational
InitExNc: allnc alpha^((Pvar alpha)alpha^ -->
                    exnc alpha^0 (Pvar alpha)alpha^0)
```

Since usage of the elimination axiom of the existential quantifier is complex, there is the command

```
(by-assume ASSUMPTION VAR NAME).
```

With this instruction the new variable `VAR`, which fulfils the existential statement `ASSUMPTION` $=: \exists_x A(x)$, is introduced and the formula $A($`VAR`$)$ is saved in the context with name `NAME`.
We demonstrate the usage of `by-assume` by proving $\forall_n(P(n) \to Q(n)) \to \exists_n P(n) \to \exists_n Q(n)$. Hence we enter

```
(set-goal "all n(P n -> Q n) -> exd n P n -> exd n Q n")
(assume "assumption1" "assumption2")
```

and the output is

```
 ok, we now have the new goal

  assumption1:all n(P n -> Q n)
  assumption2:exd n P n
-----------------------------------------------------------------
?_2:exd n Q n
```

Here we use the `by-assume` command with

```
(by-assume "assumption2" "n" "assumption2Inst").
```

Minlog returns

```
ok, we now have the new goal

  assumption1:all n(P n -> Q n)
  n  assumption2Inst:P n
--------------------------------------------------------------
?_5:exd n Q n
```

We see that `assumption2` has been dropped. But it still exists and one could use it. To prove the formula `exd n Q n` we use the introduction axiom of `EqD` by writing `(intro 0 (pt "n"))` and Minlog shows

```
ok, ?_5 can be obtained from

  assumption1:all n(P n -> Q n)
  n  assumption2Inst:P n
--------------------------------------------------------------
?_6:Q n
```

With `(use "assumption1")` and `(use "assumption2Inst")` we finishes the proof.
In contrast to `Ex` and `And` the disjunction has five variations, because the type of `OrU` is not the unit type but the Boolean algebra.

```
 OrD with content of type ysum
InlOrD: Pvar1 -> Pvar1 ord Pvar2
InrOrD: Pvar2 -> Pvar1 ord Pvar2
OrR with content of type uysum
InlOrR: Pvar1 --> Pvar1 orr Pvar2
InrOrR: Pvar2 -> Pvar1 orr Pvar2
OrL with content of type ysumu
InlOrL: Pvar1 -> Pvar1 orl Pvar2
InrOrL: Pvar2 --> Pvar1 orl Pvar2
OrU with content of type boole
InlOrU: Pvar1 --> Pvar1 oru Pvar2
InrOrU: Pvar2 --> Pvar1 oru Pvar2
OrNc non-computational
InlOrNc: Pvar1 -> Pvar1 ornc Pvar2
InrOrNc: Pvar2 -> Pvar1 ornc Pvar2
```

The axioms of the disjuctions can be applied straightforwardly so there are no special commands for the disjuction.
For each of these inductively defined predicates there are also interactive versions `andi`, `exi` and `ori`. These are not new inductively defined predicates but the

system takes the suitable version.
For example, if we want to prove the formula

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

we can do this by entering

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

Minlog recognizes that `B` and `C` could be computational and therefore it replaces `andi` by `andd`:

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

The proof of this formula in Minlog is left as an exercise.

# 4 Terms

## 4.1 define Command

Expressions can be abbreviated by the `define` command. These expressions do not have to be terms. With `define` one can also save formulas, types or just strings. The instruction has the form

```
(define STRING EXPR).
```

Instead of `STRING` one writes an arbitrary string and thus ensures that a later usage of `STRING` will be replaced by `EXPR`.
We will use the `define` command to abbreviate terms. For example, if we want to define the natural number 4, we can do this by

```
(define four (pt "Succ(Succ(Succ(Succ 0)))"))
```

but after entering `(pp four)` Minlog displays `4` and not `Succ(Succ(Succ(Succ 0)))`, since the decimal representation of natural numbers is already implemented. The extracted term of a proof will become long. Therefore the `define` command ist very useful.
But if one would like to use some formulas or types several times, it can be reasonable to set them with the `define` command. Examples for this are

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

or

```
(define sequences (py "nat=>alpha")).
```

It should be noted that one can overwrite defined strings whitout any problems and Minlog not even returns a warning.

## 4.2 Program Constants

Program constants or programmable constants are an important part of terms in Minlog. There are two steps to introduce program constants in Minlog. First one declares the name `NAME` and the type `TYPE` of the program constant by the command

```
(add-program-constant NAME TYPE).
```

We take the addition on the natural numbers as example. In the file `nat.scm` it is declared by

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

In the second step one adds the computation rules of the program constant by the command

```
(add-computation-rule TERM1 TERM2).
```

A computation rule has the form

$$D\vec{P} = M$$

where $\vec{P}$ is a list of patterns built from dictinct variables and constants, and $M$ is a term with no more free variables than those in $\vec{P}$. In the instruction above `TERM1` should be replaced by $D\vec{P}$ and `TERM2` by $M$. In the case of the addition on the natural numbers we write

```
(add-computation-rule (pt "NatPlus n Zero") (pt "n"))
(add-computation-rule (pt "NatPlus n (Succ m)")
                      (pt "Succ (NatPlus n m)")).
```

There is also an abbreviation by the command

```
(add-computation-rules
 "NatPlus n Zero" "n"
 "NatPlus n (Succ m)" "Succ(NatPlus n m)").
```

This command does the same as the two commands above but it is shorter, since one does not have to write `pt` and `add-computation-rule` for each computation rule.
If we want to see the computation rules of a program constant, we have the command

```
(display-pconst LIST)
```

which shows us the rules for the program constants in the List `LIST`. In our case entering `(display-pconst "NatPlus")` after loading `nat.scm` leads to the output

```
NatPlus
  comprules
0 n+0 n
1 n+Succ m Succ(n+m)
  rewrules
0 0+n n
1 Succ n+m Succ(n+m)
2 n+(m+l) n+m+l
```

We also see the rewrite rules of the program constant, which we consider later. To remove the program constant `NAME` we can use the instruction

```
(remove-program-constant NAME).
```

Minlog removes the name of the program constant and deletes each computation rule of it. But one should be careful with a deleted program constant. The theorems about the deleted program constant are not removed. Therefore, if a new program constant with the same name is introduced, Minlog assumes that the theorems of the old program constant also holds for the new one. But obviously this does not have to be true.

## 4.3 Examples of Program Constants

The boolean operators `andb`, `orb` and `impb` are program constants which often occur. They have the names `AndConst`, `OrConst` and `ImpConst` and also the infix notation is declared in Minlog. The computation rules are the following:

```
AndConst
  comprules
0 True andb boole^ boole^
1 boole^ andb True boole^
2 False andb boole^ False
3 boole^ andb False False
OrConst
  comprules
0 True orb boole^ True
1 boole^ orb True True
2 False orb boole^ boole^
3 boole^ orb False boole^
ImpConst
  comprules
0 False impb boole^ True
1 True impb boole^ boole^
2 boole^ impb True True
```

Another important program constant is the recursion operator. In Minlog it is denoted by

```
(Rec ALG=>TYPE).
```

Strictly speaking, this is the recursion operator from the algebra ALG into the typ TYPE. In the next section we see a term with the recursion operator. At this point we would like to consider the type of the recursion operator. The command

```
(term-to-type TERM)
```

gives the type of an general term TERM back. To see the type of the recursion operator form the natural numbers into a type alpha we write

```
(pp (term-to-type (pt "(Rec nat=>alpha)")))
```

and we get the expected output:

```
nat=>alpha=>(nat=>alpha=>alpha)=>alpha
```

The last example in this section is the decidable equality for objects of a finitary algebra. The decidable equality for a finitary algebra is automatically implemented when the algebra is defined. It is denoted by = and the computation rules are so deeply implemented in Minlog that we can not see them explicitly.

## 4.4 Abstraction and Application

In Minlog the $\lambda$ abstraction of a variable is denoted is denoted by putting these variables in square brackets and separated by commas in front of the term. The application of a term N to a term M is just denoted by M N. To show this in detail we define the addition of the natural numbers as a term of type $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ by using the recursion operator:

```
(define Plus (pt "[n,m](Rec nat=>nat) n m ([n0,n1]Succ n1)"))
```

If we would like to apply an already defined term TERM1 to another term TERM2, we can use the command

```
(make-term-in-app-form TERM2 TERM1).
```

For example the application of 1 to the term Plus can be entered by

```
(make-term-in-app-form Plus (pt "1")).
```

We also can normalize this term with nt, which is discussed two sections later, by entering

```
(pp (nt (make-term-in-app-form Plus (pt "1"))))
```

and get just Succ as output. This is consistent with the fact that addition with 1 gives the successor.
The abstraction of a variable VAR from a defined term TERM is implemented by the command

```
(make-term-in-abst-form VAR TERM).
```

So, if we enter something like the instruction (pp (make-term-in-abst-form (pv "n") (pt "n1"))), the output is [n]n+1.

## 4.5 Boolean Terms as Formulas

In Minlog a boolean term `b` can be also identified with the formula `b eqd True`. So Minlog easily understands commands like

```
(set-goal "all boole1,boole2(boole1 andb boole2 -> boole1)").
```

The identification of a boolean term with a formula is done by the theorems `EqDTrueToAtom` und `AtomToEqDTrue`, which are saved in Minlog by default:

```
AtomToEqDTrue all boole^(boole^ -> boole^ eqd True)
EqDTrueToAtom all boole^(boole^ eqd True -> boole^)
```

While working with boolean variables as formulas one often needs the theorem

```
Truth T
```

which says `T eqd T`. With this theorem one easily can prove boolean terms which normalizes to `T`.

## 4.6 Normalisation

In Minlog two terms are considered equal if they have a common reduct w.r.t. the computation and rewrite rules. This equality is also implemented in Minlog and in this chapter we take a look at the conversion rules in Minlog.
Program constants are defined by their computation rules. In the system the two sides of the rules are automatically identified. The computation rules are also saved as theorems. The `X`-th computation rule of the program constant `NAME` has the name `NAMEXCompRule`. This is one reason why the rules are numbered in the output of `display-pconst`. The computation rules do not appear in the output of `search-about`. But if one would like to see these rules, too, one can expand `search-about` by the string `'all`. So we can explicitly search for the computation rules of boolean operators by the command

```
(search-about 'all "CompRule" "Const").
```

The output is the following list:

```
Theorems with name containing CompRule and Const
NegConst1CompRule
negb False eqd True
NegConst0CompRule
negb True eqd False
OrConst3CompRule
all boole^ (boole^ orb False)eqd boole^
OrConst2CompRule
all boole^ (False orb boole^)eqd boole^
OrConst1CompRule
all boole^ (boole^ orb True)eqd True
OrConst0CompRule
```

```
all boole^ (True orb boole^)eqd True
ImpConst2CompRule
all boole^ (boole^ impb True)eqd True
ImpConst1CompRule
all boole^ (True impb boole^)eqd boole^
ImpConst0CompRule
all boole^ (False impb boole^)eqd True
AndConst3CompRule
all boole^ (boole^ andb False)eqd False
AndConst2CompRule
all boole^ (False andb boole^)eqd False
AndConst1CompRule
all boole^ (boole^ andb True)eqd boole^
AndConst0CompRule
all boole^ (True andb boole^)eqd boole^
No global assumptions with name containing CompRule and Const
```

These theorems can be used with the `simp` command.
Often one would like to normalize a term `TERM` completely. In Minlog one does this with the instruction

```
(nt TERM).
```

Of course, this is not always possible and therefore this instruction could lead to an infinite loop, which had to be stoped manually by the user.
In section 4.4 we defined the term `PLUS` and now we apply this term to two natural numbers and normalize it. So we enter

```
(define 7+2 (pt "([n,m](Rec nat=>nat) n m ([n0,n1]Succ n1))7 2"))
(pp (nt 7+2))
```

and indeed the output is `9`.
Normalization in a proof is also possible with the command

```
(ng NAME).
```

Each term in the assumption with name `NAME` is normalized by this instruction. If `NAME` is replaced by `#t`, each term in the goal formula is normalized and if one just writes `(ng)`, each term in the goal and in the context is normalized.
Since the normalisation algorithm does not need to terminate, it is sometimes better just to do $\beta$ and $\eta$ conversion. With

```
(term-to-beta-eta-nf TERM)
```

the term `TERM` is normalized referring to $\beta$ and $\eta$ conversion. As example the output of `(pp (term-to-beta-eta-nf (pt "([n]n+2)1")))` is `1+2`.
It is also possible to add new rewrite rules, which are used by the normalisation with `nt` or `ng`. By entering the command

```
(add-rewrite-rule TERM1 TERM2)
```

Minlog adds the global assumption that TERM1 and TERM2 are equal. If one uses `add-rewrite-rule` after the proof that TERM1 is equal to TERM2 this rewrite rule is added as a theorem and is not added as an assumption. In both cases this rule is always used by the commands `nt` and `ng` and appears in the output of `display-pconst`. We have already seen this in the output of `(display-pconst "NatPlus")` after loading `nat.scm`:

```
NatPlus
  comprules
0 n+0 n
1 n+Succ m Succ(n+m)
  rewrules
0 0+n n
1 Succ n+m Succ(n+m)
2 n+(m+l) n+m+l
```

Similar to the computation rules the X-th rewrite rule is saved with the name `NAMEXRewRule`.

## 4.7 The Extracted Term

After proving a formula in Minlog one can access the computational content of the proof by the command

```
(proof-to-extracted-term).
```

As an example we prove that for every even natural number $n$ there exists a natural number $m$ such that $m + m = n$ holds. Here we use the decorated version of the predicate `EvenI`, which is implemented by

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

To prove the statement above we set

```
(set-goal "allnc n(EvenI n -> exl m m+m=n)").
```

Here we use `exl`, since the equality `m+m=n` does not have any computational content, and we will use `n` only non-computationally, therefore it is bounded with an nc quantifier.
To prove the goal formula we first move with `(assume "n" "EvenIn")` the variable `n` and the premise into the context and we enter `(elim "EvenIn")` to use the elimination axiom of `EvenI`. For each introduction axiom of `EvenI` we get two new goal formulas:

```
ok, ?_2 can be obtained from
```

```
  {n}  EvenIn:EvenI n
------------------------------------------------------------
?_4:allnc n(EvenI n -> exl m m+m=n -> exl m m+m=Succ(Succ n))
```

```
  {n}  EvenIn:EvenI n
------------------------------------------------------------
?_3:exl m m+m=0
```

We prove the goal formula `exl m m+m=0` by using the introduction axiom of `ExL` with the term `0`. So we enter `(intro 0 (pt "0"))` and prove the equality `0+0=0` with `(use "Truth")`. For proving the goal formula `?_4` we first move with `(assume "n1" "EvenIn1" "IH")` everthing into the context. From the induction hypotheses `IH` we get `m` with `m+m=n1` by the command `(by-assume "IH" "m" "IHInst")`. For proving the goal formula `exl m m+m=Succ(Succ n)` we first enter `(intro 0 (pt "m+1"))`, which leads to the output

```
 ok, ?_9 can be obtained from

  {n}  EvenIn:EvenI n
  {n1}  EvenIn1:EvenI n1
  m  IHInst:m+m=n1
------------------------------------------------------------
?_10:m+1+(m+1)=Succ(Succ n1)
```

After normalization by `(ng)` the new goal is exactly `IHInst`. So we finish the proof with `(use "IHInst")`.
Now we get the extracted term by the command

```
(define eterm (proof-to-extracted-term)).
```

The output after `(pp eterm)` is hardly readable:

```
[algEvenI3985]
 (Rec algEvenI=>nat)algEvenI3985(([n^3986]n^3986)0)
 ([algEvenI3991,n3989]
   ([n3988,(nat=>nat)_3987](nat=>nat)_3987 n3988)n3989
   ([m]([n^3990]n^3990)(m+1)))
```

But the normalized form of it is more readable. Therefore we enter `(pp (nt eterm))` and get

```
[algEvenI0](Rec algEvenI=>nat)algEvenI0 0([algEvenI1]Succ)
```

back. It is always recommended to normalize an extracted term. We see that there is no variable for the algebra `algEvenI` declared, which expands the extracted term. So we define `f` as a variable of type `algEvenI`. The new normalized term is

```
[f0](Rec algEvenI=>nat)f0 0([f1]Succ)
```

The occurrence of the recursions operator matches with the fact that we used the elimination axiom of `EvenI` exactly once. The type of the extracted term `algEvenI=>nat` is also plausible, since for a witness that `n` is even we get a natural number `m` with `m+m=n`.

# 5 Totality

## 5.1 Implementation of Totality

To each algebra `ALG`, which is defined in Minlog, one can add the totality predicate by the instruction

```
(add-totality ALG).
```

The newly added predicate has the name `TotalAlg`, where the first letter of `ALG` is a capital letter.
In `list.scm` the totality of lists are introduced. With `(display-idpc "TotalList")` we take a look at the introduction axiom:

```
TotalList with content of type list
TotalListNil: TotalList(Nil alpha)
TotalListCons: allnc alpha^(
 Total alpha^ ->
 allnc (list alpha)^0(
  TotalList(list alpha)^0 -> TotalList(alpha^ ::(list alpha)^0)))
```

We see that the premise of `TotalListCons` is the totality of `alpha`. This is the absolute totality. A generalisation of absolute totality is relative totality. In Minlog it can be introduced by

```
(add-rtotality ALG).
```

It is denoted by `RTotalALG`. We take a look at relative totality of `list` by the instruction `(display-idpc "RTotalList")` and get the output

```
RTotalList with content of type list
RTotalListNil: (RTotalList [...])(Nil alpha)
RTotalListCons: allnc alpha^(
 (Pvar alpha)_356 alpha^ ->
 allnc (list alpha)^0(
  (RTotalList [...])(list alpha)^0 ->
  (RTotalList [...])
  (alpha^ ::(list alpha)^0)))
```

For better readability

```
(RTotalList (cterm (alpha^1) (Pvar alpha)_356 alpha^1))
```

is replaced by (RTotalList [...]). As we see, relative totality has a type variable alpha and also a predicate variable (Pvar alpha)_356. Compared to absolute totality, relative totality has this predicate variable instead of totality of alpha in the premise of TotalListCons. Relative totality is a generalization of absolute totality. Each totality of another type is replaced by a predicate variable. If one inserts for each predicate variable in the relative totality predicate the predicate $\mu_X(\forall_x Xx)$, one gets structural totality, which is also a variation of totality.

## 5.2 Implicit Representation of Totality

The reader may have seen in previous sections that some variable names in the output of Minlog are equipped with ^. Now we see why. For each quantification over a variable without ^ Minlog adds implicitly the assumption that this variable is total. Therefore expressions like $\forall_x A$ and $\forall_x^{nc} A$ are abbreviations for $\forall_{\hat{x}}^{nc}(\texttt{Total}\ \hat{x} \to A)$ and $\forall_{\hat{x}}^{nc}(\texttt{Total}\ \hat{x} \to^{nc} A)$. The character ^ after the variable name indicates that the variable does not have to be total. In Minlog this abbreviation is introduced by two theorems

```
AllncTotalIntro allnc alpha^(Total alpha^ --> (Pvar alpha)alpha^)
 -> allnc alpha(Pvar alpha)alpha
AllTotalIntro allnc alpha^(Total alpha^ -> (Pvar alpha)alpha^)
 -> all alpha(Pvar alpha)alpha
```

and it is eliminated by

```
AllncTotalElim allnc alpha(Pvar alpha)alpha ->
 allnc alpha^(Total alpha^ --> (Pvar alpha)alpha^)
AllTotalElim all alpha(Pvar alpha)alpha ->
 allnc alpha^(Total alpha^ -> (Pvar alpha)alpha^)
```

For each version of the existential quantifier there are the introduction axioms for the totality abbreviation

```
ExDTotalIntro
 exr alpha^(Total alpha^ andd (Pvar alpha)alpha^) ->
 exd alpha(Pvar alpha)alpha
ExLTotalIntro
 exr alpha^(Total alpha^ andl (Pvar alpha)^' alpha^) ->
 exl alpha(Pvar alpha)^' alpha
ExRTotalIntro
 exr alpha^(TotalNc alpha^ andr (Pvar alpha)alpha^) ->
 exr alpha(Pvar alpha)alpha
ExNcTotalIntro
exnc alpha^(Total alpha^ andnc (Pvar alpha)alpha^) ->
exnc alpha(Pvar alpha)alpha
```

and also the elimination axioms of the totality abbreviation.

```
ExNcTotalElim
 exnc alpha (Pvar alpha)alpha ->
 exnc alpha^(Total alpha^ andnc  (Pvar alpha)alpha^)
ExRTotalElim
 exr alpha (Pvar alpha)alpha ->
 exr alpha^(TotalMR alpha^ alpha^ andr (Pvar alpha)alpha^)
ExLTotalElim
 exl alpha (Pvar alpha)^ alpha ->
 exr alpha^(Total alpha^ andl (Pvar alpha)^ alpha^)
ExDTotalElim
 exd alpha (Pvar alpha)alpha ->
 exr alpha^(Total alpha^ andd (Pvar alpha)alpha^)
```

Of course, these theorems only can be used if the corresponding (absolute) totality predicate is defined. But even if the totality of the corresponding type is not defined, one can use variables with and without `^`. As long as the totality of this type is not defined, there is no semantical difference between `x` and `x^`. Nevertheless for example, the system never allows a specialisation of a formula $\forall_x A$ to a variable $\hat{x}$.

## 5.3 Totality of Program Constants

For proving that a program constant `PCONST` is total there is the command

```
(set-totality-goal PCONST).
```

The computer generates as goal formula the totality of this program constant. In the file `nat.scm` the totality of the addition on the natural numbers is proven. This is done by

```
(set-totality-goal "NatPlus").
```

Minlog gives the detailed goal

```
---------------------------------------------------------------
?_1:allnc n^(TotalNat n^ ->
             allnc n^0(TotalNat n^0 -> TotalNat(n^ +n^0)))
```

back. To prove this we move with `(assume "n^" "Tn" "m^" "Tm")` everything into the context and prove `TotalNat(n^ +m^)` by the elimination axiom for `Tm`. Therefore we enter `(elim "Tm")` and get the output

```
ok, ?_2 can be obtained from

  {n^}  Tn:TotalNat n^
  {m^}  Tm:TotalNat m^
---------------------------------------------------------------
?_4:allnc n^0(TotalNat n^0 ->
```

```
                TotalNat(n^ +n^0) -> TotalNat(n^ +Succ n^0))


  {n^}  Tn:TotalNat n^
  {m^}  Tm:TotalNat m^
-----------------------------------------------------------------
?_3:TotalNat(n^ +0)
```

After normalisation with `(ng #t)` the first goal is exactly the assumption `Tn`. So we prove it with `(use "Tn")`. For the second goal we enter first the instruction `(assume "l^" "Tl" "IH")` to move the premises and the variables into the context. This leads to the output:

```
ok, we are back to goal

  {n^}  Tn:TotalNat n^
  {m^}  Tm:TotalNat m^
  {l^}  Tl:TotalNat l^
  IH:TotalNat(n^ +l^)
-----------------------------------------------------------------
?_5:TotalNat(n^ +Succ l^)
```

When we normalise the goal formula with `(ng #t)`, we get the new goal `TotalNat(Succ(n^ +l^))`. Here we use the totality of the successor function, which is exactly the second introduction axiom of `TotalNat`. So we enter `(use "TotalNatSucc")`. Minlog requires a proof that the argument of `Succ` is total, i.e. `TotalNat(n^ +l^)`. This is done by `(use "IH")` and the proof is finished.
After proving the totality of a program constant `PCONST` one can save the totality by

```
(save-totality).
```

This instruction saves the totality of `PCONST` as a theorem with name `PCONSTTotal`.

## 5.4 Totality of Boolean Terms

For total boolean terms the logical connectives $\to, \wedge$ and $\vee$ are equvialent to the boolean once `impb`, `andb` and `orb`. Especially for `andb` this fact is also implemented in Minlog. Therefore one can use the command `(split)` also to prove a formula of the form `a andb b`, where `a` and `b` are total boolean terms. In the other direction, one can prove `a` or `b` directly by using the assumption `a andb b`. For example the goal

```
all boole1,boole2(boole1 andb boole2 -> boole1)
```

is easily proven by

```
(assume "boole1" "boole2" "assumption")
(use "assumption").
```

To demonstrate the usage of `(split)` we prove the goal

```
all boole1,boole2(boole1 andb boole2 -> boole2 andb boole1).
```

In Minlog this is done by entering

```
(assume "boole1" "boole2" "assumption")
(split)
(use "assumption")
(use "assumption").
```

Here it is important that both variables are total. The statement is also true if only one of both variables is total. But then `(split)` and `(use "assumption")` does not lead to a correct proof and one has to prove it in another way.
For total variables of a finitary algebra the decidable equality is equivalent to the Leibniz equality. In Minlog this is not automatically implemented such that one has to prove it for each algebra individually. In the library file this is done for some algebras. In `nat.scm` the theorem `NatEqToEqD` and in `list.scm` the theorems `ListBooleEqToEqD`, `ListNatEqToEqD` and `ListListNatEqToEqD` are proven.
If for an finitary algebra `ALG` the theorem

```
all a,b(a = b -> a eqd b)
```

is saved with the name `ALGEqToEqD`, then we also can use formulas of the form `T1 = T2` instead of `T1 eqd T2` as arguments of `simp`, if `T1` and `T2` are total terms with type `ALG`. If such a theorem is not saved, the theorem `ALGEqToEqD` becomes a global assumption by the usage of simp as above.

## 5.5 Induction

To use the abbreviation of totality efficiently there is the command

```
(ind).
```

This command can be used if the goal formula has the form $\forall_x A(x)$. Minlog applies the elimination axiom of the totality of $x$ to the predicate $\{x|A(x)\}$ and the premises of this axiom become the new goal formulas. An extended command of `ind` is

```
(ind TERM)
```

for a total term `TERM`. This instruction can be applied to each goal formula $A($`TERM`$)$. In this case the elimination axiom of the totality of `TERM` is applied to $\{x|A(x)\}$.
As an example we prove, that each total natural number is even or odd. Therefore we load the file `nat.scm` and define `EvenI` and `OddI` as in section 2.3 and 2.4. To set the goal we enter

```
(set-goal "all n (EvenI n ord OddI n)").
```

Of course, here we use the computational disjunction `OrD` since `EvenI` and `OddI` have computational content. After setting the goal formula we directly use induction on `n`. Therefore we enter `(ind)` and get

```
ok, ?_1 can be obtained from

  n3987
-----------------------------------------------------------------
?_3:all n(EvenI n ord OddI n
          -> EvenI(Succ n) ord OddI(Succ n))


  n3987
-----------------------------------------------------------------
?_2:EvenI 0 ord OddI 0
```

back. The goal `?_2` holdes because of `EvenI 0`. So we enter `(intro 0)` twice and this goal is proven. For the goal `?_3` we first move with `(assume "n" "IH")` the variable and the premise into the context and then use the elimination axiom of `OrD` by the command `(elim "IH")`. Then Minlog requires twice a proof of `EvenI(Succ n) ord OddI(Succ n)` once with the assumption `EvenI n` and once with the assumption `OddI n`.

```
ok, ?_5 can be obtained from

  n3990  n  IH:EvenI n ord OddI n
-----------------------------------------------------------------
?_7:OddI n -> EvenI(Succ n) ord OddI(Succ n)


  n3990  n  IH:EvenI n ord OddI n
-----------------------------------------------------------------
?_6:EvenI n -> EvenI(Succ n) ord OddI(Succ n)
```

If `EvenI n` holds, we prove `Odd(Succ n)`. Therefore we enter `(assume "Evenn")` followed by `(intro 1)`. The formula `OddI(Succ n)` can be proven by using the elimination axiom of `(EvenI n)` by `(elim "Evenn")`. The output of Minlog is

```
ok, ?_9 can be obtained from

  n3996  n  IH:EvenI n ord OddI n
  Evenn:EvenI n
-----------------------------------------------------------------
?_11:all n(EvenI n -> OddI(Succ n)
           -> OddI(Succ(Succ(Succ n))))


  n3996  n  IH:EvenI n ord OddI n
```

```
  Evenn:EvenI n
-----------------------------------------------------------------
?_10:OddI 1
```

`OddI 1` is the first introduction axiom of `OddI` and so we prove it with `(intro 1)`. The goal `?_11` follows from the second introduction axiom of `OddI`. We just enter `(assume "n1" "Evenn1")` and `(use "GenOddI")` and it is proven. So the case `EvenI n` is done. If `OddI n` holds, the proof is done analogously:

```
(assume "Oddn")
(intro 0)
(elim "Oddn")
(use "GenEvenI")
(intro 0)
(assume "n1" "Oddn1")
(use "GenEvenI")
```

As normalized extracted term we get

```
[n0]
 (Rec nat=>algEvenI ysum algOddI)n0
      ((InL algEvenI algOddI)CInitEvenI)
 ([n1,(algEvenI ysum algOddI)_2]
   [if (algEvenI ysum algOddI)_2
     ([algEvenI3]
      (InR algOddI algEvenI)
      ((Rec algEvenI=>algOddI)algEvenI3 CInitOddI
       ([n4,algEvenI5]CGenOddI(Succ n4))))
     ([algOddI3]
      (InL algEvenI algOddI)
      ((Rec algOddI=>algEvenI)algOddI3(CGenEvenI 0 CInitEvenI)
       ([n4,algOddI5]CGenEvenI(Succ n4))))])
```

The first recursion operator corresponds to the elimination axiom of the totality, which we used with `(ind)`. The other two recursion operators come from the elimination of `EvenI n` and `OddI n`. The elimination of `EvenI n ord OddI n` with the command `(elim "IH")` provides a recursion operator on the sum of types. In the normalized extraced term this occurs as `if`. In Minlog `if` denotes the case operator. For a sum of types the case operator and the recursion operator are program constants with the same computational rules and Minlog uses the case operator, since it is simpler. In the next section we take a deeper look on the case operator.

## 5.6 Case Distinction

In a proof by induction it often occurs that one does not need the induction hypothesis or there is not even an induction hypothesis. For example, we have the last case for the boolean totality. The elimination axiom of the boolean totality is $\forall_b(\mathbf{T}_{\mathbb{B}} b \to P\mathtt{tt} \to P\mathtt{ff} \to Pb)$. As we see, this is just case distinction by the two cases $b = \mathtt{tt}$ and $b = \mathtt{ff}$. But also for total natural numbers there are formulas which can be proven just by distinction between the zero case and the successor case. As example we take the (modified) predecessor function on the natural numbers, which is a program constant `Pred` given by the computation rules `Pred` $0 := 0$ and `Pred` $Sn := n$. One way of proving that this program constant is total, i.e. $\forall_n(\mathbf{T}_{\mathbb{N}} n \to \mathbf{T}_{\mathbb{N}}(\mathtt{Pred}\, n))$, is by using the elimination axiom of totality. In Minlog this proof is done as follows:

```
(set-totality-goal "Pred")
(use "AllTotalElim")
(ind)
(intro 0)
(use "AllTotalIntro")
(assume "n^" "Tn^" "Spam")
(use "Tn^")
```

After entering `(use "AllTotalIntro")` we get the output

```
 ok, ?_4 can be obtained from

  n4149
-----------------------------------------------------------------
?_5:allnc n^(TotalNat n^ -> TotalNat(Pred n^)
             -> TotalNat(Pred(Succ n^)))
```

but the premise `TotalNat(Pred n^)` is not used in the proof. As extraced term we have

```
[n0][if n0 0 ([n1]n1)]
```

which do not have a recursion operator, although we have used the command `(ind)`. Therefore we rather expect an extracted term like

```
[n0](Rec nat=>nat) n0 0 ([n1,n2]n1).
```

But, since we have not used the induction hypothesis, the bounded variable `n2` does not appear anywhere else. In such cases the recursion operator can be replaced by the simpler case operator. In Minlog this is done automatically by the normalisation of such a term. The case operator is denoted by `[if ...]`, where `...` stands for the arguments of the case operator.
If one just would like to do case distinction on a total variable, there is the instruction

```
(cases),
```

which are analogously used as `(ind)`. We have also the expansion

```
(cases TERM)
```

of the `cases` command to do case distinction on a total term `TERM`. So we can also prove the totality of `Pred` with `cases`:

```
(set-totality-goal "Pred")
(use "AllTotalElim")
(cases)
(intro 0)
(use "AllTotalIntro")
(assume "n^" "Tn^")
(use "Tn^")
```

The proof is almost similar but after `(use "AllTotalIntro")` we have the output

```
ok, ?_4 can be obtained from

  n4312
---------------------------------------------------------------
?_5:allnc n^(TotalNat n^ -> TotalNat(Pred(Succ n^)))
```

Here the induction hypothesis `TotalNat(Pred n^)` does not occur as premise. One often uses `cases` for a variable whose algebra just have constructor types of the form $\kappa(\xi) = \vec{\sigma} \to \xi$. This is the case for the boolean algebra and the sum algebra. But also for the type product the `case` command is useful, even though there is only one case. As an educational example we define the program constant `sort`, which rearranges a pair of natural numbers such that the smaller number is on the left-hand side. In Minlog we do this by the instruction

```
(add-program-constant "sort"
    (py "(nat yprod nat)=>(nat yprod nat)"))
(add-computation-rules
  "sort (n pair m)" "[if (m<n) (m pair n) (n pair m)]")
```

We declare `x` as a variable of type `nat yprod nat` and then we enter the goal formula

```
(set-goal "all x lft(sort x) <= rht(sort x)")
```

Here we directly use the command `(cases)` and Minlog shows all possible cases how `x` can be built. There is only the case `x = n pair m` for total natural numbers `n` and `m`. Therefore the output is

```
 ok, ?_1 can be obtained from

  x4406
---------------------------------------------------------------
?_2:all n,n0 lft(sort(n pair n0))<=rht(sort(n pair n0))
```

and we have reached that `x` is decomposed. Now we enter `(assume "n" "m")` and normalize the goal formula by `(ng)`. Since `sort` is defined by case distinction on `m<n`, we also use case distinction on `m<n` for the proof and write therefore `(cases (pt "m<n"))`. So we get two new goals:

```
 ok, ?_4 can be obtained from

  x4414  n  m
-----------------------------------------------------------------
?_6:(m<n -> F) ->
    lft[if False (m pair n) (n pair m)]
    <=rht[if False (m pair n) (n pair m)]



  x4414  n  m
-----------------------------------------------------------------
?_5:m<n -> lft[if True (m pair n) (n pair m)]
           <=rht[if True (m pair n) (n pair m)]
```

In each case the term `m<n` was already replaced. After `(assume "case1")` and `ng` we get for the first case:

```
ok, the normalized goal is

  x4430  n  m  case1:m<n
-----------------------------------------------------------------
?_8:m<=n
```

We prove this with the theorem `NatLtToLe: all n,m(n<m -> n<=m)`, which is implemented in `nat.scm`. So we enter `(use "NatLtToLe")` followed by `(use "case1")`. The second case is analogously proven. Here we need the theorem `NatNotLtToLe: all n,m((n<m -> F) -> m<=n)` to get `n<=m` from the assumption `m<n -> F`.

# References


[1] Ulrich Berger, Kenji Miyamoto, Helmut Schwichtenberg, and Monika Seisenberger. Minlog — a tool for program extraction supporting algebras and coalgebras. In *CALCO*, pages 393–399, 2011.

[2] Kenji Miyamoto. The Minlog System, 2017. [Online; accessed 20-November-2017].

[3] Helmut Schwichtenberg and Stanley S. Wainer. *Proofs and Computations*. Perspectives in Logic. Association for Symbolic Logic and Cambridge University Press, 2012.

[4] Franziskus Wiesnet. Kostruktive Analysis mit exakten reellen Zahlen. Master's thesis, Ludwig Maximilians University, 2017.