Matita Tutorial

Andrea Asperti

August 27, 2007

Contents

Chapter 1

Getting started

1.1 Natural Numbers

Natural numbers are the smallest set generated by a constant O and a successor function S. Sets of this kind, freely generated by a finite number of *constructors*, are known as *inductive types*.
The definition of natural numbers in Matita reads as follows:

```
inductive nat : Set \def
  | O : nat
  | S : nat \to nat.
```

By this syntax, we declare we are defining an inductive type of name `nat` (which is a `Set`), built up from the two constructors `O` of type `nat` and `S` of type `nat` → `nat`.

Let us now define a simple function over natural numbers. The sum of two natural numbers n and m may be defined as follows: if n is O then the sum is m, otherwise, if n is the successor of some natural number p, then the sum of n and m is equal to the successor of the sum of p and m. In this way, we have reduced the computation of the sum of two natural numbers to a similar problem, but on smaller input values. This is an example of a *recursive* definition, that is a definition of a function in terms of the function itself.

In Matita, the previous definition of the sum over natural numbers would be written in the following way:

```
let rec plus n m \def
```

```
match n with
[ O \Rightarrow m
| (S p) \Rightarrow S (plus p m) ].
```

The `let rec` construct is used to declare a recursive function with name `plus`. The body of the function follows the `\def` keyword: it starts with a pattern-matching operation over the input variable n. Since n is a natural number, it is either `O` or `(S p)` for some integer p; in the first case the result is m, otherwise the result is the successor of the recursive call `(plus p m)`.

Let us now prove our first theorem. The first thing to do is to give a name to the theorem. The choice of the name is not entirely negligible, since it is the only way we shall have to refer to it (e.g. for later use inside different proofs). We shall come back to this problem in section **??**; for the moment we merely suggest to give a name that refers to the statement, it is easy to remember and possibly also easy to guess.

In the next example, we are going to prove that for any n we have $n = O + n$; we give to thie result the name `plus_O_n`:

```
theorem plus_O_n: \forall n:nat. n = plus O n.
```

The first thing to note is the syntax for binders; in particular, we work in a typed framework and the quantified variable should be followed by the declaration of its type: `n:nat` in the previous case. Secondly, the body of the quantifier is introduced by a "dot" (a notation borrowed from the λ-calculus of Church); the scope of the binder is thus the expression following the dot.

When you enter the previous theorem declaration in Matita, the system automatically starts a new proof-session, opening a goal window in the right upper corner. Initially, there is only one goal to prove, that is the given statement. The proof essentially proceed in a bottom-up fashion, by giving proof-command to the system, called *tactics*. After executing these commands we shall be typically left with some sub-goals to prove, and we shall go on in this way until all subgoals will be finally closed.

All proofs typically start assuming the hypothesis of the statement. This operation is performed by the `intros` tactic. After executing `intros` we are left to prove $n = plus\ O\ n$ in a *context* where we have assumed $n : nat$.

$n : nat$

$n = plus\ O\ n$

The horizontal line divides the context from the goal.
The definition of plus, is algorithmic: so we may *reduce* the current goal, using e.g. the `simplify` command (see section **??** for other kinds of reductions). In particular, $plus\ O\ n \rightsquigarrow n$, so after simplification we are left with the goal.

$$n : nat$$
$$\text{---}$$
$$n = n$$

This is obviously true by *reflexivity* of equality, and not surprsingly we conclude the proof with an invocation of the `reflexivity` tactic. Once the proof is finished, i.e. we have no open goal left, we should still invoke the `qed` command, that re-check the proof and saves it in the current environment.
The complete proof reads as follows:

```
theorem plus_O_n: \forall n:nat. n = O+n.
  intros.simplify.reflexivity.
qed.
```

Let us now try to prove that for any n, $n = n+O$ (we have not proven yet the commutativity of *plus*, so the result is not entirely trivial). We would expect to proceed as in the previous proof, so we start with `intros`, and then try to `simplify`. But (maybe) here comes a surprise: the expression $n = n + O$ does not reduce to n. The reason is that the function `plus` works by case analysis on the first argument, and if this argument is not an instance of a constructor, no reduction is possible.
So, what should we do?
In this case, we must proceed by induction on n. In Matita, the name of the tactic invoking the induction principle for a variable of a given inductive type is `elim`. So, instead of `simplify` we type `elim n`:

```
theorem plus_n_O: \forall n:nat. n = n+O.
  intros. elim n.
```

At this point, we are left with two goals: we must prove that the goal is true when n is O (base case), and we must prove that, supposing the statement holds for n, that is $n = n + O$, it still holds for the successor of n, that is $Sn = (Sn) + O$ (inductive case).

goal 1	*goal* 2
$n : nat$	$n : nat$
$\overline{\qquad\qquad\qquad}$	$n1 : nat$
$O = plus\ O\ O$	$H : n1 = plus\ n1\ O$
	$\overline{\qquad\qquad\qquad\qquad}$
	$S\ n1 = plus\ (S\ n1)\ O$

The first goal is easy: we close it with `simplify` and `reflexivity`. For the second goal, after simplification we get:

$n : nat$
$n1 : nat$
$n1 = plus\ n1\ O$
$\overline{\qquad\qquad\qquad\qquad}$
$S\ n1 = S(plus\ n1\ O)$

Now we have to use the induction hypothesis H, *rewriting* $(plus\ n1\ O)$ with $n1$. This is done by invoking the tactic `rewrite < H`, where H must be the proof of an equality, and $<$ gives the orientation (`rewrite > H` would rewrite $n1$ with $plus\ n1\ O$).
After the rewriting step, we may conclude the proof with reflexivity. The whole proof runs as follows:

```
theorem plus_O_n: \forall n:nat. n = O+n.
  intros.elim n.
    simplify.reflexivity.
    simplify.rewrite < H.reflexivity.
qed.
```

In a very (sic!) similar way we prove that forall n and m, $plus\ n\ (S\ m) = S\ (plus\ n\ m)$:

```
theorem plus_n_Sm : \forall n,m:nat. S (n+m) = n+(S m).
  intros.elim n.
    simplify.reflexivity.
    simplify.rewrite < H.reflexivity.
qed.
```

We are now ready to prove the commutativity property. As usual, we begin assuming the hypothesis, and then prooceed by induction on n. At

this point we are left with two subgoals that, after simplification, look as follow:

goal 1	*goal* 2
$n : nat$	$n : nat$
$m : nat$	$m : nat$
———	$n1 : nat$
$m = plus\ m\ O$	$H : plus\ n1\ m = plus\ m\ n1$
	———
	$S\ (plus\ n1\ m) = plus\ m\ (S\ n1)$

So, we may close the former with `plus_n_O` and the latter rewriting $plus\ n1\ m$ with $plus\ m\ n1$ (by H), and then using `plus_n_Sm`. The tactic that allows to use an already proved result is `apply foo`, where `foo` is the name of the result. The whole proof is thus:

```
theorem sym_plus: \forall n,m:nat. n+m = m+n.
  intros.elim n.
    simplify.apply plus_n_O.
    simplify.rewrite > H.apply plus_n_Sm.
qed.
```

The tactics `intros`, `simplify`, `elim`, `rewrite` and `apply` form a basic set of elementary tactics that is already complete for the purpose of proving theorems. Many other tactics are in fact particular cases of the previous ones or can be essentially expressed as a suitable composition of these tactics. For instance, the `reflexivity` tactic literally amounts to apply the proof `refl_eq` of the reflexivity principle; the `exact` tactic is similar to `apply` but it also presumes to close the current goal; the useful tactic `assumption` iterate the application of exact on all the hypothesis in the current context, and so on.

1.2 Double induction

To have a gist of the power of the tools we already have, let us prove the following double induction principle for natural numbers, that we shall frequently use in the following sections.

```
theorem nat_elim2 :
 \forall R:nat \to nat \to Prop.
  (\forall n:nat. R O n)
  \to (\forall n:nat. R (S n) O)
  \to (\forall n,m:nat. R n m \to R (S n) (S m))
  \to \forall n,m:nat. R n m.
```

Let us observe, first of all, the type of R. *Prop* is the universe of all (definable) Propositions, and R is a function that, given two natural numbers, gives back a proposition; in other words, R is just a *binary relation* over natural numbers. The stement says that if the relation holds along the two axes $R\ O\ n$ and $R\ m\ O$, and all diagonals, then it holds everywhere.

In order to prove `nat_elim2` we start adding to the context the *first five* hypothesis by means of the command `intros 5` (the numeric parameter express the number of premisses to shift in the context; writing `intro` is the same as `intros 1`), getting the goal $\forall m.R\ n\ m$. We now proceed by induction on n, that produces two subgoals:

goal 1

$R : nat \to nat \to Prop$
$H : \forall n : nat.R\ O\ n$
$H1 : \forall n : nat.R\ (Sn)\ O$
$H2 : \forall n, m : nat.$
$\quad R\ n\ m \to R\ (S\ n)\ (S\ m)$
$n : nat$
$m : nat$

$\forall m : nat.R\ O\ m$

goal 2

$R : nat \to nat \to Prop$
$H : \forall n : nat.R\ O\ n$
$H1 : \forall n : nat.R\ (Sn)\ O$
$H2 : \forall n, m : nat.$
$\quad R\ n\ m \to R\ (S\ n)\ (S\ m)$
$n : nat$
$n1 : nat$
$H3 : \forall m : nat.R\ n1\ m$
$m : nat$

$R\ (S\ n1)\ m$

The first goal is subsumed by H. For the second one it is enough to reason by *cases* on m, that is essentially analogous to induction, but for the fact that we renounce to the induction hypothesis. In particular, in our case, the command `cases m` transform the second goal in the two subgoals below:

goal 2.1

$R : nat \to nat \to Prop$
$H : \forall n : nat.R\ O\ n$
$H1 : \forall n : nat.R\ (Sn)\ O$
$H2 : \forall n, m : nat.$
$\quad R\ n\ m \to R\ (S\ n)\ (S\ m)$
$n : nat$
$n1 : nat$
$H3 : \forall m : nat.R\ n1\ m$
$m : nat$

$R\ (S\ n1)\ O$

goal 2.2

$R : nat \to nat \to Prop$
$H : \forall n : nat.R\ O\ n$
$H1 : \forall n : nat.R\ (Sn)\ O$
$H2 : \forall n, m : nat.$
$\quad R\ n\ m \to R\ (S\ n)\ (S\ m)$
$n : nat$
$n1 : nat$
$H3 : \forall m : nat.R\ n1\ m$
$m : nat$
$n2 : nat$

$R\ (S\ n1)\ (Sn2)$

The first goal is subsumed by $H1$, while for the second one we have just to apply $H2$ and $H3$. The full script is:

```
theorem nat_elim2 :
 \forall R:nat \to nat \to Prop.
  (\forall n:nat. R O n)
  \to (\forall n:nat. R (S n) O)
  \to (\forall n,m:nat. R n m \to R (S n) (S m))
  \to \forall n,m:nat. R n m.
intros 5;elim n
  [ apply H
  | cases m
    [ apply H1
    | apply H2. apply H3 ] ]
qed.
```

1.3 Negation and Discrimination

Two major axioms of Peano axiomatization of natural numbers are those stating the inequality, for any x, between O and $S\ x$, and the injectivity of

S. More generally, given any inductive type, we expect to be able to prove that all constructors are injective and distinguishable from each other.

The statement $\forall x.Not\ (O = (Sx))$ is our first example involving *negation*. In the logical framework of Matita, negation is not considered as a primitive connective, but is instead defined in terms of the absurdity proposition $False$. Explicitly, we may conclude $Not\ P$, if and only if assuming P leads to an asburdity, that is $P \to False$.

What about False? The only logical principle related to $False$ is the ancient property expressed by the latin motto *ex falso quodlibet*, that is, everything may be deduced under a false assumption. Formally, this is expressed by the following property that, for the moment, we may assume to be a primitive constant of the system:

$$False\_ind : \forall P : Prop.False \to P$$

As for injectivity/separability of constructors, Matita provides a tactic *destruct* to this aim. The tactic expects in input a term of type $e_1 = e_2$, and recursively compare e_1 and e_2, skipping common constructors (injectivity) and halting on subexpressions differing on their leftmost outermost symbols. If any of these correponding symbols are two different constructors the tactics automatically closes the goal having obtained a contradiction. Otherwise, it adds to the context all new equalities between the different subformulae in corresponding positions.

Luckily, the tactic is much simpler to use than to explain. Let us see a couple of examples.

```
theorem not_eq_O_S: \forall n:nat. Not (O= S n).
```

We start with `intros`, then `unfold Not` to change the goal into $O = S\ n \to False$, and `intro` again, to push $O = S\ n$ into the context. We are left with the goal

$n : nat$
$H : O = S\ n$

$False$

and to close it, it is enough to call `destruct H`.
Let us look at the injectivity of the successor function:

```
theorem inj_S: \forall n,m:nat. S n = S m \to n = m.
```

After `intros` we have the goal

$n : nat$
$H : S\ n = S\ m$

$n = m$

In this case, `destruct H` add to the context a new hypothesis $n = m$, and we may hence close the goal with `assumption`.

Chapter 2

Defining properties

So far, the only property we have been dealing with has been equality that we have essentially assumed as a primitive notion[1] of Matita, governed via rewriting and reflexivity (prove simmetry and transitivity as an exercise).

Our next step is to define a binary predicate `le n m` asserting that n is less or equal to m. As we shall see, we have a lot of different ways to do it.

Let us start from the following mathematical definition: n is less or equal to m if and only if it exists p such $n + p = m$. We have essentially two different, almost equivalent, way to encode such a predicate in our logical framework: as a definition or as an inductive type.
In the first case, we would write:

```
definition le1: nat \to nat \to Prop \def
\lambda n,m.\exist p:nat.n+p = m.
```

However, an experienced *Matita* user would probably prefer to transform the previous definition of `le` into the smallest property induced by an integer p and a proof that $n + p = m$, namely:

```
inductive le2 (n,m:nat) : Prop \def
le_witness : \forall p:nat.n+p = m \to le2 n m.
```

`le_witness` is the user-defined name of the costructor; it could be changed with any other name without affecting the semantics of the definition. Note

[1] See more in Section ??

that the two parameters n and m are also parameters for the constructor: in other words the full type of `le_witness` is

$$\forall n, m, p : nat. n + p = m \to le2\ n\ m$$

It may be instructive to formally prove the equivalence between the two previous definitions. Let us look at the first implication:

```
lemma le1_to_le2:
  \forall n,m.le1 n m \to le2 n m.
```

After introducing the hypothesis, we are left with the following goal:

$n : nat$
$m : nat$
$H : le1\ n\ m$

$le\ n\ m$

First of all we must unfold the defintion of `le1` in the hypothesis H, that is done with the command:

```
unfold le1 in H.
```

The execution of the previous command transforms the hypothesis H into the formula $\exists p : nat. n + p = m$. The next step is to use the $\exists$-elimination rule, that just amounts to type `elim H`. At this point we are left with the goal

$n : nat$
$m : nat$
$H : \exists p : nat. n + p = m$
$p : nat$
$H1 : n + p = m$

$divides\ n\ m$

and we may close the proof with `exact (le_witness n m p H1)`.
The full script is hence:

```
lemma le1_to_le: \forall n,m.le1 n m \to le n m.
intros.
unfold divides1 in H.
elim H.
exact (le_witness n m p H1).
qed.
```

The other direction is *very* similar:

```
lemma le2_to_le1: \forall n,m.le n m \to divides1 n m.
intros (n m H).
elim H.
exact (ex_intro ? ? n2 H1).
qed.
```

In the previous proof, $n2$ is the witness obtained eliminating H, while `ex_intro` is a (library) constant corresponding to the logical rule of $\exists$-introduction, with type

$$\forall A : Type.\forall P : A \to Prop.\forall x : A.Px \to \exists x : A.Px$$

Let us remark that this second proof is slightly more compact than the previous one, not requiring a preliminary unfolding of the definition of *le* before using (eliminating) it. This is one of the practical reason for preferring `le2` over `le1`; more generally, as we shall see in section ??, the existential quantifier is just a particular case of inductive type, and the direct definition of a property as an inductive types does usually allow a simpler and more direct destructuration of the corresponding definition.

2.1 Recursive properties

The previous definition of *le* relies on the definition of the sum, that is not very elegant. An alternative way to look at the less or equal relation is as the smallest relation R being *reflexive* and such that $R\ n\ m$ implies $R\ n\ (S\ m)$. This notion is precisely captured by the following inductive type:

```
inductive le (n:nat) : nat \to Prop \def
  | le_n : le n n
  | le_S : \forall m:nat. le n m \to le n (S m).
```

Again, let us prove the equivalence between this definition and, say, the definition of `le2`.

```
lemma le2_to_le: \forall n,m:nat. le2 n m \to le n m .
```

As usual, we start with `intros`; then we eliminate the hypothesis `le2 n m`, by typying `elim H 1`, obtaining the following goal:

$n : nat$
$m : nat$
$H : le2\ n\ m$
$n2 : nat$

$n + n2 = m \to divides\ n\ m$

Note the "1" at the end of the `elim` invocation. By default, the `elim` tactic automatically executes `intros` as a final operation, hence also the premise $n + n2 = m$ would have been shifted from the conclusion into the context. The numeric parameter "1" is actually interpeted as an argument for `intros`, that is as the number of new hypothesis to be added to the context.

At this point, we would expect to proceed by induction on $n2$; unfortunately, we would soon get stuck (check it as an exercise). This is a typical situation: we need to prove a given statement, but in order to prove it by induction, we need a to prove a stronger one, since otherwise we would not be able to properly use the induction hypothesis. In our case, the proposition we want to prove is not

$$n + n2 = m \to divides\ n\ m$$

where m would be a fixed parameter, but

$$\forall m.n + n2 = m \to divides\ n\ m$$

Hence, we need to *generalize* our goal with respect to m, that is done by invoking `generalize in match m` (the syntax of Matita is here a bit awkward; we plan to change it in the future).

The rest of the script merely uses results we already know, and the reader should have no problem to follow its execution. Here is the full proof:

```
lemma le2_to_le: \forall n,m:nat. le2 n m \to le n m .
intros.
elim H 1.
generalize in match m.
elim n2
  [rewrite < plus_n_O in H1.
   rewrite < H1.
```

```
   apply le_n
  |rewrite < plus_n_Sm in H2.
   rewrite < H2.
   apply le_S.
   apply H1.
   reflexivity.
  ]
qed.
```

The converse is equally easy.

```
lemma le_to_le2: \forall n,m:nat. le n m \to le2 n m.
intros.
elim H
  [apply (le_witness ? ? O).
   rewrite < plus_n_O.
   reflexivity
  |elim H2.
   apply (le_witness ? ? (S n2)).
   rewrite < plus_n_Sm.
   apply eq_f.
   assumption
  ]
qed.
```

2.2 Prop vs. Bool

The definition of `le` of the previous section does not give an effective way to *decide* if n is less or equal to m. What we are looking for is a binary boolean function returning $true$ if $n \leq m$, and $false$ otherwise. Such a computable function is usually called a *decision procedure*. A property admitting a decision procedure is called *recursive*, or *decidable*[2].

First of all, let us define the type of booleans; it is the smallest type just containing two elements true and false:

[2]One of the major mathematical results of the 20-th century, and one of the starting point of Computability Theory, has been the discovery that not all arithmetical properties are decidables

```
inductive bool : Set \def
  | true : bool
  | false : bool.
```

Then, we may define our decision procedure for the less or equal relation in then following way:

```
let rec leb n m \def
  match n with
  [ O \Rightarrow true
  | (S p) \Rightarrow
    match m with
    [ O \Rightarrow false
    | (S q) \Rightarrow leb p q]].
```

As soon as you have a decision procedure, it is convenient to define a corresponding elimination principle, that allows to relate in a very general way the boolean function (in this case `leb`) to its propositional counterpart (`le`). In our case, the elimination principle says that for all natural numbers n and m and any predicate P over booleans, provided we may prove $(P\ true)$ under the assumption $(n \leq m)$, and $(P\ false)$ under the assumption $n \not\leq m$, then we may conclude $(P\ (leb\ n\ m))$:

$$(n \leq m \to (P\ true)) \to (n \not\leq m \to (P\ false)) \to P(leb\ n\ m)$$

The proof requires three small lemmas, left as an exercise for the reader:

```
theorem not_le_Sn_O:
  \forall n:nat. S n \nleq O.
theorem le_S_S:
  \forall n,m:nat. n \leq m \to S n \leq S m.
theorem le_S_S_to_le :
  \forall n,m:nat. S n \leq S m \to n \leq m.
```

One we have thee thre lemmas above, the proof of `leb_elim` is more or less straightforward:

```
theorem leb_elim: \forall n,m:nat. \forall P:bool \to Prop.
(n \leq m \to (P true)) \to (n \nleq m \to (P false)) \to
```

```
P (leb n m).
apply nat_elim2; intros; simplify
  [apply H.apply le_O_n
  |apply H1.apply not_le_Sn_O.
  |apply H;intros
    [apply H1.apply le_S_S.assumption.
    |apply H2.unfold Not.intros.
     apply H3.apply le_S_S_to_le.assumption
    ]
  ]
qed.
```

Let us just remark the use of the separator ";". The semicolon is *distributive*: in a configuration of the kind $c_1; c_2$ the latter command c_2 will be applied to all goals opened by c_1. In many cases, the semicolon allows to sensibly shorten the lenght of the scripts, but it can be always avoided, and its use is essentially a matter of taste.

The second remark is about the management of *negation*. By definition $Not\ P$ is equivalent to $P \to False$. Hence, having the goal

$n : nat$
$m : nat$
$H : le1\ n\ m$

$le\ n\ m$

In order to appreciate the power of `leb_elim` let us prove that forall n and m, $leb\ n\ m = true \to n \leq m$, and viceversa (in general, it is better to avoid to state formal theorems using the if-and-only-if connective, since they are cumbersome to use).

The formal statement is the following

```
theorem leb_true_to_le:
  \forall n,m:nat. leb n m = true \to n \leq m.
```

We start shifting n and m into the context, hence apply `leb_elim` and `intros` again, on all subgoals. At this point we are left with two subgoals:

goal 1	*goal* 2
$n : nat$	$n : nat$
$m : nat$	$m : nat$
$H : n \leq m$	$H : n \not\leq m$
$H1 : true = true$	$H1 : false = true$
$n \leq m$	$n \leq m$

The first goal is trivally closed by `assumption`. As for the second goal, it contains the contradictory hypothesis $false = true$, so again the goal is true since *ex falso quodlibet*. This is the complete proof:

```
intros 2.apply leb_elim;intros
  [assumption
  |apply False_ind.apply not_eq_true_false.
   apply sym_eq.assumption
  ]
qed.
```

The proof of the converse, that is

```
theorem le_to_leb_true_to:
  \forall n,m:nat. n \leq m \to leb n m = true.
```

is even simpler, and is left as exercise for the reader.

Chapter 3

A non trivial example

In a way similar to `plus` we may define the product of two natural numbers:

```
let rec times n m \def
 match n with
 [ O \Rightarrow O
 | (S p) \Rightarrow m+(times p m) ].
```

Our next step is to define a binary predicate `divides n m` asserting that n is a divisor of m. As exlplained in the previous section, we may either use a defintion or an inductive type. In the first case, we would write:

```
definition divides1: nat \to nat \to Prop \def
\lambda n,m.\exist p:nat.m = times n p.
```

However, as we have seen, it is preferable to define *divides* as the smallest property induced by a "witness" p and a proof that $m = np$, namely:

```
inductive divides (n,m:nat) : Prop \def
witness : \forall p:nat.m = times n p \to divides n m.
```

Let us now address the decidability of *divides*, i.e. the problem to define an algorithmic boolean function that taken in input two integers n and m returns $true$ if n divides m, and $false$ otherwise.

A convenient way to proceed is to address the slightly more general problem to compute the *modulus* (mod) of two numbers (i.e. the rest of an integewr division); obviously n *divides* m if and only if $m \ mod \ n = 0$.

The natural way to define `mod` as a recursive function would look something like the following:

```
let rec mod m n: nat \def
match (leb (S m) n) with
[ true \Rightarrow m
| false \Rightarrow mod (m-n) n
]
```

The problem with this definition is that the calculus is based on a particular kind or recursion that must be guaranteed to be well-founded (never diverge)[1]. This is typically the case when the recursive parameter "decreases" in the recursive call. Unfortunately, there is no trivial way, for the system, to understand that $m-n$ is "smaller' than m w.r.t. some well founded ordering (minus is a user defined operation!). What the system is able to understand is essentially a syntactic ordering, based on the structure of inductive data: for instance n is "smaller" of $S\ n$.

A simple but effective approach to this problem is that of defining an auxiliary function accepting in input an extra-parameter t providing an upper bound to the complexity of the function, and then recurring on such an argument. For instance, in the case of the modulus, we may eventually complete the computation of $mod\ m\ n$ in less than m steps. So, we define:

```
let rec mod_aux t m n: nat \def
match (leb (S m) n) with
[ true \Rightarrow m
| false \Rightarrow
    match t with
    [O \Rightarrow m
       (* if t is large enough this case never happens *)
    |(S t1) \Rightarrow mod_aux t1 (m-n) n
    ]
].

definition mod : nat \to nat \to nat \def
\lambda m,n.mod_aux m n n.
```

[1] In a programming language, a typing judgement $e : T$, should actually be understodd as asserting that *provided e is defined*, e has Type T. So, for instance, we may easily imagine a undefined (divergent) expression inhabiting a void type: at the meta-level, this would easily lead to logical inconsistencies.

When we define a function having several cases as the previous one, it is good practice to specify the behavour with a distinct lemma for each possible branch. In the case of the previous function, we have the following relevant cases, whose proof is more or less straightforward. The only novelty is the use of the `change` tactic, that allows to replace a goal with an equivalent one (up to convertibility). Sometimes, the heuristic used by the `simplify` tactic simplifies too much, and it turns out to be convenient to give the system the expected transformation in an explicit way.

```
lemma O_to_mod_aux: \forall m,n. mod_aux O m n = m.
intros.
simplify.elim (leb (S m) n);reflexivity.
qed.

lemma lt_to_mod_aux:
\forall t,m,n. m < n \to mod_aux (S t) m n = m.
intros.
change with
( match (leb (S m) n) with
  [ true \Rightarrow m
  | false \Rightarrow mod_aux t (m-n) n] = m).
rewrite > (le_to_leb_true ? ? H).
reflexivity.
qed.

lemma le_to_mod_aux:
\forall t,m,n.
  n \le m \to mod_aux (S t) m n = mod_aux t (m-n) n.
intros.
change with
(match (leb (S m) n) with
[ true \Rightarrow m
| false \Rightarrow mod_aux t (m-n) n] = mod_aux t (m-n) n).
apply (leb_elim (S m) n);intro
  [apply False_ind.apply (le_to_not_lt  ? ? H).apply H1
  |reflexivity
  ]
qed.
```

In order to understand the use of the previous lemmas, let us try to prove a simple property of the modulus operation, namely that for any positive n, $mmodn < n$.

The proof has the following structure:

```
theorem lt_mod_aux_m_m:
\forall n. O < n \to
 \forall t,m. m \leq t \to (mod_aux t m n) < n.
intros 3.
elim t
  [rewrite > O_to_mod_aux.
   apply (le_n_O_elim ? H1).
   assumption
  |elim (decidable_lt m n)
    [rewrite > lt_to_mod_aux[assumption|assumption]
    |rewrite > le_to_mod_aux
      [apply H1.
       ...
      |apply not_lt_to_le.
       assumption
      ]
    ]
  ]
qed.
```

Mimiking the definition of `mod_aux`, the proof proceeds by induction on the recursive parameter t. The case $t = 0$ is closed by `O_to_mod_aux`; in case $t = Sn1$, we distinguish two more case according if $m < n$ or not. In the former case, we use $lt_to_mod_aux$ while in the latter we use `le_to_mod_aux` and the inductive hypothesis `H1`. The dots correspond to a trivial but formally cumbersome fragment of the proof. Indeed, after applying the inductive hypothesis `H1` we are left with the following goal:

$n : nat$
$H : O < n$
$t : nat$
$n1 : nat$
$H1 : \forall m : nat.m \leq n1 \rightarrow mod\_aux\ n1\ m\ n < n$
$m : nat$
$H2 : m \leq S\ n1$
$H3 : m \not< n$

$m - n \leq n1$

The proof requires several elementary arithmetical results, but is not particualry informative, so we shall skip it here.

Having defined the modulus, we define

```
definition divides_b : nat \to nat \to bool \def
\lambda n,m :nat. (eqb (m \mod n) O).
```

Bibliography

]