# nat.scm

Simone Testino

November 2023

## Introduction

The library `nat.scm` is the one containing all axioms, theorems and definitions on natural numbers. I will first start by presenting the definition of natural numbers by the function `Succ` together with the constant `Zero`. After that I will present the so-called `program-constant`, those will be the definitions necessary to use natural numbers as we are used to in mathematical practice.

`add-algs`

```
(add-algs "nat" '("Zero" "nat") '("Succ" "nat=>nat"))
(add-var-name "n" "m" "l" (py "nat"))
;l instead of k, which will be an int
```

We start the program with the already familiar (tutor: 6.4) command `(add-algs "nat" '("Zero" "nat") '("Succ" "nat=>nat"))` which creates an algebra with constructors `Zero` and `Succ`. The algebra will then be closed under the successor function and have zero as its starting element. The second line adds the default variables `n`, `m` and `l` as natural numbers.

# Program constants

As announced we then proceed by adding program constants to the library, those will be the functions, predicates and relations we are already familiar with, I will analyse those signed by `;;*` at the end of the line.

```
;; Program constants.
(add-program-constant "NatPlus" (py "nat=>nat=>nat")) ;;*
(add-program-constant "NatTimes" (py "nat=>nat=>nat"))
(add-program-constant "NatLt" (py "nat=>nat=>boole"))
(add-program-constant "NatLe" (py "nat=>nat=>boole")) ;;*
(add-program-constant "Pred" (py "nat=>nat"))
(add-program-constant "NatMinus" (py "nat=>nat=>nat"))
(add-program-constant "NatMax" (py "nat=>nat=>nat"))
(add-program-constant "NatMin" (py "nat=>nat=>nat"))
(add-program-constant "AllBNat" (py "nat=>(nat=>boole)=>boole")) ;;*
(add-program-constant "ExBNat" (py "nat=>(nat=>boole)=>boole"))
(add-program-constant "NatLeast" (py "nat=>(nat=>boole)=>nat")) ;;*
(add-program-constant "NatLeastUp" (py "nat=>nat=>(nat=>boole)=>nat"))
```

The command `add-program-constant` (or for short, `apc`, also in tutor 6.4) is the one that introduces the program constant by defining its type (both domain and codomain. As we know, in order to define a function, predicate or relation, it is not enough to define its domain and codomain, instead one should give some computational rules, what follows contains the computational rules for all of the marked functions and relations. Those will be sometimes followed by brief and simple lemmas which we are already very familiar with in mathematical practice.

## NatPlus

I use this program constant as an example of what stated above, therefore I will shortly repeat the essential passages together with the corresponding instances.

### add-program-constant

Here we first introduce the domain and codomain of the function, we notice that, as expected, + is a function that takes two objects of type `nat` and returns a third object of type `nat`.

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

### add-display

In order to have a notation looking more familiar, we can modify the notation of `NatPlus` to `+`. In order to do so, we may simply use the following command:

```
(add-display (py "nat") (dc "NatPlus" "+" 'add-op))
```

### add-computation-rules

The following are the computation rules that we expect for +, in fact they precisely correspond to the two familiar axioms of Peano Arithmetic concerning +: (i) $\forall_x(x+0=0)$, (ii) $\forall_{x,y}(x+S(y)=S(x+y))$.

```
(add-computation-rules
"n+0" "n"
"n+Succ m" "Succ(n+m)")
```

### NatPlusComm

Among the different theorems that we can prove from the definition of + together with the former definitions, I present the commutativity of +, claiming $\forall_{n,m} n+m=m+n$.
First I prove the claim in the familiar mathematical vocabulary one can then compare it with the proof of the lemma on `Minlog`.
*Claim.* $\forall_{n,m\in\mathbb{N}} n+m=m+n$.
*Proof.* Consider a natural number $n$, then by induction we wish to prove that the statement above holds for all $m$, in order to do so, we have two claims: $n+0=0+n$ and that $n+m=m+n \rightarrow n+m+1=m+1+n$. In order to prove the induction start we use the computation rule stating that both sides are equal $n$. In order to prove the induction step, we first take one $m$ and assume the induction hypothesis, then notice that by the second computation rule we conclude the proof.

```
;; NatPlusComm
(set-goal "all n,m n+m=m+n")
(assume "n")
(ind)
(use "Truth")
(assume "m" "IH")
(use "IH")
```

## NatLe

Just like above we implement some more familiar notation for a relation operator:

```
(add-display (py "boole") (dc "NatLe" "<=" 'rel-op))
```

Then consider the computation rules, those are again defined inductively, meaning that first the `0` case gets defined and then the successor case with `Succ n`.

```
;; For NatLe
(add-computation-rules
"0<=n" "True"
"Succ n<=0" "False"
"Succ n<=Succ m" "n<=m")
```

As expected, we see that the `0` case gives always `True` if it is on the right of the relation and `False` if it is on the left; also we have $n+1 \leq m+1$ iff $n \leq m$ which clearly holds for our intuition of $\leq$ and defines it in the successor case where nor of the two is 0.
There are now two lemmas on $\leq$ to prove, I proceed like above.

`NatLeTrans`

Since there are some commands that might be new to the reader, here I introduce those before starting to prove the claim.
The command `(strip)` is equivalent to a repetition of `(assume "...")` for all you can assume, the example we are about to see is of how `(strip)` transforms the goal `all m,l(0 <= m -> m <= l -> 0 <= l)` in the assumptions `0 <= m`, `m <= l` and the goal `0 <= l`. The command `(use "Truth")` will then prove the goal, this command concludes trivial arguments like the one above.
The command `(cases)`, as one images, divides the proof into two possible cases, one when an object is assumed to be `0` and the other when the object is assumed to be a successor.
The assertion `"Absurd"` allows us to drop those assumptions that are always false, for example the goal `Succ n <= 0 -> 0 <= l -> Succ n <= l` can be simplified to `0 <= l -> Succ n <= l` since a false antecedent proves the implication tautologically true.
The assertion `EfAtom` is just like the familiar `Efq` but only for atomic formulae, e.g. we'll use it in the form `F -> Succ n <= l`.
Since the proof will be a little more intricate than the previous one, I will state the proof more like in a Minlog style, so that the Minlog proof can be better understood.
*Claim.* $\forall_{n,m,l}(n \leq (sm \to m \leq l \to n \leq l)$
*Proof.* We first clearly need to proceed by induction, hence we consider have now the two goals $\phi(0)$ and $\phi(n) \to \phi(n+1)$ for $\phi$ the transitivity of $\leq$.
First consider $\phi(0) = \forall_{m,l}(0 \leq m \to m \leq l \to 0 \leq l)$. First instantiate $m$ and $l$ and then assume bot $0 \leq m$ and $m \leq l$ (all in the command `(strip)`), finally notice that $0 \leq l$ trivially holds, hence the claim is proved.
Now consider the claim $\phi(n) \to \phi(n+1)$. Instantiate $n$ and assume $\phi(n)$, then consider the two cases (i) where $m = 0$ (ii) where $m$ is a successor.
Consider the case where $m = 0$, namely: $n+1 \leq 0 \to 0 \leq l \to n+1 \leq l$. Notice that the first antecedent is always false, hence the material implication is always true, then conclude $0 \leq l \to n+1 \leq l$ (`(assume "Absurd")`. Then assume the antecedent $(0 \leq l)$ and in order to prove the consequent, derive it from the absurd assumption `Succ n <= 0` using `EfAtom`.
Consider now (ii.i), for $m$ a successor (starts from `(assume "m")`, now again, consider two cases (ii.i) where $l = 0$ and (ii.ii) where $l$ is a successor.
Consider the case $l = 0$, namely $n+1 \leq m+1 \to m+1 \leq 0 \to n+1 \leq 0$, notice that both $m+1 \leq 0$ and $n+1 \leq 0$ cannot be the case, hence antecedent and consequent of the last material implication are always false, hence the implication is true.
Consider the case (ii.ii), namely $n+1 \leq m+1 \to m+1 \leq l+1 \to n+1 \leq l+1$ and this corresponds to $\phi(n)$ that we assumed, hence the claim is proved.
All four goals (i), (ii), (ii.i) and (ii.ii) are achieved, hence the induction step is also concluded.

```
;; NatLeTrans
(set-goal "all n,m,l(n<=m -> m<=l -> n<=l)")
(ind)
(strip)
(use "Truth")
(assume "n" "IH")
(cases)
(assume "l" "Absurd" "H1")
(use "EfAtom")
```

```
(use "Absurd")
(assume "m")
(cases)
(assume "H1" "Absurd")
(use "Absurd")
(use "IH")
;; Proof finished.
;; (cdp)
(save "NatLeTrans")
```

`NatLeCases`

Two other lemmas of `nat.scm` will be usefull in the next proof, I present here the claims.
The lemma `"NatLeAntiSym"` states `all n,m(n<=m -> m<=n -> n=m)`, namely that $n \leq m$, $m \leq n \vdash m = n$.
The lemma `"NatNotLtToLe"` states `all n,m((n<m -> F) -> m<=n)`, namely that $\neg n < m \vdash m \leq n$.
*Claim.* $\forall_{n,m}(n \leq m \to (n < m \to \varphi) \to (n = m \to \varphi) \to \varphi)$
*Proof.* First instantiate $n$ and $m$ and assume the antecedent $n \leq m$, now consider two cases (i) $n < m$ and (ii) $\neg n < m$.
Consider the case where $n < m$, then we assume the antecedents and get the assumptions $n < m \to \varphi$ and ($n = m \to \varphi$ and the goal $\varphi$ which follows from the first assumption directly `(use-with "THyp" "Truth")`.
Now consider the case $\neg n < m$, again assume the antecedents and we get the assumptions: $m \leq m$ (from before), $\neg n < m$, $n = m \to \varphi$ and the goal $\varphi$, we use the last assumption and get the only goal $n = m$. We use then the antisymmetry (with `"NatLeAntiSym"`) of $\leq$ to prove that $m \leq n$ together with $n \leq m$ (which we have already) would give us the goal (with `"NatNotLtToLe"`), and we can prove $m \leq n$ from the fact that we know $\neg n < m$, then proof finished.

```
;; NatLeCases
(set-goal "all n,m(n<=m -> (n<m -> Pvar) -> (n=m -> Pvar) -> Pvar)")
(assume "n" "m" "n<=m")
(cases (pt "n<m"))
;; Case n<m
(assume "n<m" "THyp" "FHyp")
(use-with "THyp" "Truth")
;; Case n<m -> F
(assume "n<m -> F" "THyp" "FHyp")
(use "FHyp")
(use "NatLeAntiSym")
(use "n<=m")
(use "NatNotLtToLe")
(use "n<m -> F")
;; Proof finished.
;; (cdp)
(save "NatLeCases")
```

## NatLeast

This program constant takes as input a natural number `n` and a property on natural numbers `ps`, namely an element of type `nat => boole`. The output of this function is the least number less than `n` such that `ps` holds. Clearly this function is defined recursively and therefore we analyse the case of when it is applied to `0` and to a successor `Succ n`.

```
;; For NatLeast
(add-computation-rules
"NatLeast 0 ps" "0"
"NatLeast(Succ n)ps"
"[if (ps 0) 0 (Succ(NatLeast n([m]ps (Succ m))))]")
```

As we see from the computational rule, this constant is defined such that, if we take `n` to be `0`, then the result must be `0` (note that this is a non-trivial convention) and if the input is `Succ n`, then we have that if `ps` applies to `0`, then the outcome must be `0`, else one repeats recursively looking for the first successor such that `ps` holds.

## Transformation of an Injection to a Permutation

To make use of the program constants and lemmas proved, let's try to prove the following statement:
*Claim.* One can always expand an injection $f : X \to Y$ s.t. $|X| = |Y|$ to a bijection.
*1st Proof.* Let $f : X \to Y$ be an injection, then define $F : X \to Y$ s.t. $F(x) = f(x)$ and for $y \notin f(X)$, $\exists_{x \in X}(F(x) = y \wedge \forall_{y_1 \in f(X)} y \neq y_1)$.
Notice that this proof does not *construct* $F$, instead, I gave conditions that $F$ must respect and only proved that that family of functions is non-empty. Though, picking one such function in particular requires the *Axiom of Choice* and therefore the proof is to be labelled as *non-constructive*. Hence, such a proof could not be inserted in Minlog. The following proof will now be closer to what we saw up to now in Minlog, it will then be both constructive and regard natural numbers.
*2nd Proof.* Let $f : [0, n] \to Y$ be an injective function s.t. $Y \subseteq \mathbb{N}$. Let $k + 1$ be the least number s.t. $k+1 \notin f([0, n])$ (for that use `NatLeast`). Note that since $f$ is injective $k \geq n$, hence we have two cases: (i) $k = n$, in such a case, $f$ would already be bijective since $f$ injective and $|f([0, n])| = |[0, n]|$, hence take $\sigma = f$ and (ii) $k > n$. In this latter case, proceed by finite recursion by taking the least element in $[0, k]$, call it $y$ s.t. $\neg\exists_{x \in [0,n]} f(x) = y$ (again use `NatLeast`), then define $\sigma(y) = n + j$ for $j$ the step of the recursion. For a last and finite $j$ equal to $k - n$, we have constructed a bijection $\sigma : [0, k] \to [0, k]$ which can then be expanded to $\sigma_\infty : \mathbb{N} \to \mathbb{N}$ by simply extending it with $id_{\mathbb{N}}$.

## Induction

The library contains not only program constants and some lemmas strictly on them but also theorems that are true on natural numbers. In particular we know that induction is a mathematically intricate concept at the heart of natural numbers, here we examine two more intricate proofs on induction: `CVIndPvar` and `CVInd`. Those two proof differ in the use of `((Pvar nat)m)` and `ps m`, the former is a non-computable predicate variable, though the latter is an element of type `nat => bool`. Hence the difference here lays in whether a computable procedure to determine the predicate for each variable has been given or not.

### `CVIndPvar`

Just like before, I proceed by showing the some new commands and then explaining the proof in natural language.
The quantifier `allnc` that we read in the claim stands for "non-computational" all quantifier, they are used in cases where the variable on which it quantifies won't be used freely in the proof, though, logically it has no difference from the familiar `all`.
Note that the first antecedent stands for the assumption that from $\bot$ we can derive the desired statement, a weakening of the general "Ex Falso Quodlibet"
The command `(assert "...")` adds a new goal and sets it as an antecedent of the present goal.
*Claim.* $(\bot \to \forall_n(\varphi(n))) \to \forall_n(\forall_{m<n}(\varphi(m)) \to \varphi(n)) \to \forall_n \varphi(n)$.
*Proof.* First assume the antecedent (with `(assume "efq")`), then assume also $\forall_n \forall_{m<n}(\varphi(m)) \to \varphi(n)$ and keep $\forall_n \varphi(n)$ as a goal.
Now claim $\forall_{n,m}(m < n \to \varphi(m))$ (with `(assert "...")`), and proceed proving it by induction on $n$. First note that the induction start, $\forall_m(m < 0 \to \varphi(m))$, follows trivially from falsity of the antecedent $m < 0$ (with `(assume "m" "Absurd")`).
On the induction step of the claim, namely: $(\forall_{m<n} \to \varphi(m)) \to \forall_m m < n + 1 \to \varphi(m)$, simply assume both antecedents: $\forall_{m<n} \to \varphi(m)$ and $\forall_m m < n + 1$ and set the goal to $\forall_m \phi(m)$. Now consider the cases (i) $m = n$ and (ii) $m < n$, we can do that since we know that $m < n + 1$ holds `(use "m < Succ n")`.
Consider (ii) where $m < n$, then prove $\phi(m)$ thanks to $\forall_m m < n \to \varphi(m)$, the induction hypothesis.
Now consider (i), assume $m = n$ and keep the goal $\varphi(m)$, use the assumption `Prog` to get $\forall_{m<n} \varphi(m)$ and conclude the case.

Now we go back to the assumption made trough `(assert "...")`, claiming that the assertion proves our former goal, namely: $\forall_{n,m}(n < m \to \varphi(m)) \to \forall_n \varphi(n)$. After assuming the antecedent `(assume "Assertion")`, we can use it instantiating $m = n + 1$ to get the claim.

```
;; CVIndPvar
(set-goal "(F -> allnc n^(Pvar nat)n^) ->
all n(all m(m<n -> (Pvar nat)m) -> (Pvar nat)n) ->
all n (Pvar nat)n")
(assume "efq" "Prog")
(assert "all n,m(m<n -> (Pvar nat)m)")
(ind)
(assume "m" "Absurd")
(use "efq")
(use "Absurd")
(assume "n" "IHn" "m" "m<Succ n")
(use "NatLtSuccCases" (pt "m") (pt "n"))
(use "m<Succ n")
(use "IHn")
(assume "m=n")
(simp "m=n")
(use "Prog")
(use "IHn")
(assume "Assertion" "n")
(use "Assertion" (pt "Succ n"))
(use "Truth")
;; Proof finished.
;; (cdp)
(save "CVIndPvar")
```

`CVInd`

*Claim.* $\forall_\phi \forall_n(\forall_m(m < n \to \varphi(m)) \to \varphi(n)) \to \forall_n \varphi(n)$
*Proof.* Assume the antecedent and call it `Prog`, then `assert` $\forall_{n,m} m < n \to \varphi(m)$, now I have to both prove the assertion and that it implies $\forall_n \varphi(n)$, the previous goal.
In order to prove the assertion proceed by induction on $n$, goals now are induction start and induction step. Induction start has $m < 0$ as an antecedent and results therefore trivially true (with `(use "Absurd")`). For the induction step first assume the antecedents, the goal will be $\varphi(m)$, then, with `NatLtSuccCases` distinguish (i) $m = n$ and (ii) $m < n$ thanks to `(use "m < Succ n")`.

```
;; CVInd
(set-goal "all ps(all n(all m(m<n -> ps m) -> ps n) -> all n ps n)")
(assume "ps" "Prog")
(assert "all n,m(m<n -> ps m)")
(ind)
(assume "m" "Absurd")
(use "EfAtom")
(use "Absurd")
(assume "n" "IHn" "m" "m<Succ n")
(use "NatLtSuccCases" (pt "m") (pt "n"))
(use "m<Succ n")
(use "IHn")
(assume "m=n")
(simp "m=n")
(use "Prog")
(use "IHn")
(assume "Assertion" "n")
```

```
(use "Assertion" (pt "Succ n"))
(use "Truth")
;; Proof finished.
;; (cdp)
(save "CVInd")
```

## Further Useful Constants

In this last section I present a couple of program constants that have been implemented afterwards and which will be very useful for the upcoming discussions. Lastly, I encourage the reader to give a look to the proof of `NatTimesChooseNatF`, the claim will certainly result familiar, though the length of the proof couldn't fit this script. The proved claim is $\binom{m}{k} = \frac{n!}{k!\cdot(n-k)!}$

`Choose`

This program constant implements the binomial coefficients, the constant is added as a function of type `nat => nat => nat` and defined recursively.

```
;; 2023-04-16.  Choose (binomial coefficients) added
(add-program-constant "Choose" (py "nat=>nat=>nat"))
(add-computation-rules
"Choose Zero Zero" "Succ Zero"
"Choose Zero(Succ m)" "Zero"
"Choose(Succ n)Zero" "Succ Zero"
"Choose(Succ n)(Succ m)" "Choose n m+Choose n(Succ m)")
```

The `Zero` case defines $\binom{0}{0} = 1$, then we have $\binom{0}{m+1} = 0$ and $\binom{n+1}{0} = 1$, finally $\binom{n+1}{m+1} = \binom{n}{m} + \binom{n}{m+1}$. those computation rules define by induction the program constants as the binomial coefficients we are familiar with.

`NatF`

Finally consider the faculty function of type `nat => nat` defined by recursion.

```
;; 2023-03-05.  Faculty added.  Relation between Choose and NatF proved.
(add-program-constant "NatF" (py "nat=>nat"))
(add-computation-rules
"NatF Zero" "Succ Zero"
"NatF(Succ n)" "NatF n*(Succ n)")
```

The two computational rules needed to define this function are $0! = 1$ and $(n+1)! = n! \cdot (n+1)$.