# nat.scm

Simone Testino

November 2023

The library `nat.scm` is the one containing all axioms, theorems and definitions on natural numbers. In particular it is composed by:

- Definition of the Algebra
- Definitions of Program Constants
- Theorems

We will give a look of example of all of those.

# Introduction

More precisely we will give a look at:

- Definition of type `nat`
- `NatPlus`, `NatLe`, `NatLeast`, `Choose`, `NatF`
- `NatPlusComm`, `NatLeTrans`, `NatLeCases`, `CVIndPvar`, `CVInd`

Each of the theorems will come with proof in natural language and the `Minlog` code.

# Define nat

```
(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 define the type 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`.

# Program Constants: NatPlus

Program Constants require first to be added through:

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

Then, in order to have a more familiar notation we add:

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

Finally we define its meaning through computational rules:

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

# Program Constants: `NatPlus`

The computational rules of `NatPlus` together with the other assumptions of `nat.scm` enable us to make proofs, the following is a brief example:

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

# Program Constants: `NatPlus`

*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.

# NatLe

Proceed similarly and define $\leq$:

```
(add-program-constant "NatLe" (py "nat=>nat=>boole"))
(add-display (py "boole") (dc "NatLe" "<=" 'rel-op))
(add-computation-rules
"0<=n" "True"
"Succ n<=0" "False"
"Succ n<=Succ m" "n<=m")
```

We wish now to prove transitivity `NatLeTrans`.

Before starting with the proof, I list some relevant commands:

- `(strip)` will be used to assume all quantified variables and antecedents.
- `(cases)` 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
- `"EfAtom"` is just like the assumption `Efq` but only for atomic fromulae.

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
$n+1 \leq m+1 \rightarrow m+1 \leq 0 \rightarrow n+1 \leq 0$, 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
$n+1 \leq m+1 \rightarrow m+1 \leq l+1 \rightarrow n+1 \leq l+1$ 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.

# 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.* $\forall_{n,m}(n \leq m \to (n < m \to \varphi) \to (n = m \to \varphi) \to \varphi)$

# NatLeCases

*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.

# 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.

```
(add-program-constant "NatLeast" (py
"nat=>(nat=>boole)=>nat")) (add-computation-rules
"NatLeast 0 ps" "0"
"NatLeast(Succ n)ps"
"[if (ps 0) 0 (Succ(NatLeast n([m]ps (Succ m))))]")
```

# Transformation of an Injection to a Permutation

*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.

# Transformation of an Injection to a Permutation

*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

I am about to prove two lemmas 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.

Let's see ome unfamiliar commands first:
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.

*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")`.
The case (ii) is solved thanks to the induction hypothesis, for case (i), assume $m = n$ and set the goal $\varphi(m)$, through `Prog` we get the goal $\forall_{m<n} \varphi(m)$, which is again proved by the induction hypothesis.
Left to prove is only the fact that the assertion proves the previous goal, namely: $\forall_{n,m} (m < n \to \varphi(m)) \to \forall_n \varphi(n)$. Assume the assertion and use it setting $m = n + 1$ and the proof is concluded.

# Binomial Coefficients

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

```
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.

# Factorial

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

```
(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 $(n+1)! = n! \cdot (n+1)$.

The theorem `NatTimesChooseNatF` that claims: $\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$ has been proved.