Basics of CafeOBJ and Peano Style Natural Numbers

Lecture Note 01
Formal Methods (i613-0912)

Topics

- Basic concepts for modeling, specification, verification in CafeOBJ
- Basics of CafeOBJ language system: module, signature, equation, term, reduce, parse
- Specification and verification of Peano style natural numbers

Modeling, Specifying, and Verifying in CafeOBJ

1. By understanding a problem to be modeled/ specified, determine several sorts of objects (entities, data, agents, states) and operations (functions, actions, events) over them for describing the problem

2. Define the meanings/functions of the operations by declaring equations over expressions/terms composed of the operations

3. Write proof scores for properties to be verified

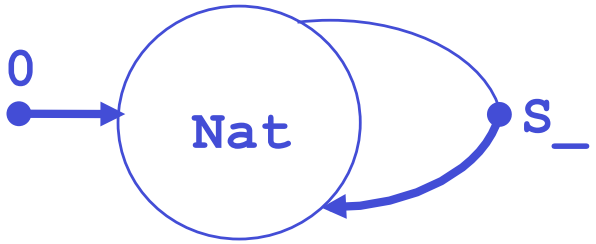
Natural Numbers -- Signature --

0 0+1 0+1+1 0+1+1+1 0+1+1+1+1 ...

0 s(0) s(s(0)) s(s(s(0))) s(s(s(s(0)))) ...

objects: Nat
operations: 0 : returns zero without arguments
s : given a natural number *n* returns the next natural number (s *n*) of *n*

```
-- sort
[ Nat ]
--constructor operators
op 0 :  -> Nat {constr}
op s_:  Nat -> Nat {constr}
```



Natural Number
-- Expressions/terms composed of operators --

1. 0 is a natural number
2. If n is natural number then (s n) is a natural number
3. An object which is to be a natural number by 1 and 2 is only a natural number

Peano's definition of natural numbers (1889), Giuseppe Peano (1858-1932)

```
Nat = {0, s(0), s(s(0)), s(s(s(0))), s(s(s(s(0)))) … }
```

```
Nat = {0, s 0, s s 0, s s s 0, s s s s 0, … }
```

Describe a concept in expressions/terms!

CafeOBJ module specifying PNAT -- Peano Style natural numbers

```
mod! PNAT {
  [ Nat ]
  op 0 : -> Nat {constr} .
  op s_ : Nat -> Nat {constr} .

  op _=_ : Nat Nat -> Bool {comm} .
  eq (N:Nat = N) = true .
  eq (0 = s(N2:Nat)) = false .
  eq (s(N1:Nat) = s(N2:Nat)) = (N1 = N2) .
}
```

Constructors (indicated by {constr}) define recursively/ inductively the set of terms which constitute a sort.

Mathematical Induction over Natural Numbers -- induced by declaration of constructors

The recursive structure defined by two constructors of sort Nat induces the following induction scheme.

Goal: Prove that a property $P(n)$ is true for any natural number $n \in \{0, s\ 0, s\ s\ 0, \ldots\}$

Induction Scheme:

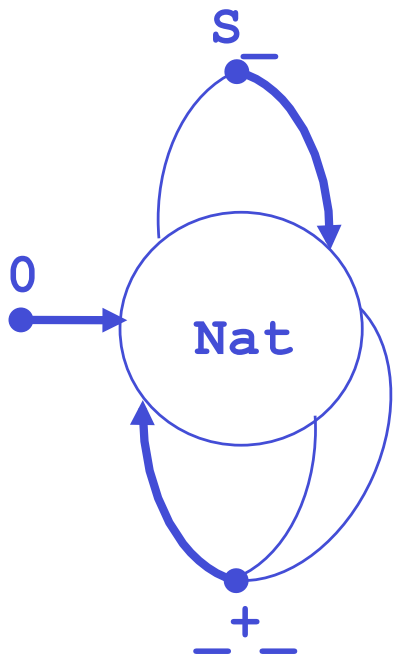
$$\frac{P(0) \qquad \forall n \in \mathbf{N}.[P(n) \Rightarrow P(s\ n)]}{\forall n \in \mathbf{N}.P(n)}$$

Concrete Procedure: (induction with respect to n)

1. Prove $P(0)$ is true
2. Assume that $P(n)$ holds, and prove that $P(s\ n)$ is true

Natural numbers with addition operation -- signature and expressions/terms

```
-- sort
[ Nat ]
-- operations
op 0 :   -> Nat {constr}
op s_:   Nat -> Nat {constr}
op _+_: Nat Nat -> Nat
```



$$\mathrm{Nat} = \{\ 0\ \} \cup \{\ s\ n \mid n \in \mathrm{Nat}\ \} \cup \{\ n1 + n2 \mid n1 \in \mathrm{Nat} \wedge n2 \in \mathrm{Nat}\ \}$$

Natural numbers with addition

-- expressions/terms composed by operators --

```
Nat = {
0, s 0, s s 0, s s s 0, ... ,
0 + 0, 0 + (s 0), 0 + (s s 0), 0 + (s s s 0), ...,
(s 0) + 0, (s 0) + (s 0), (s 0) + (s s 0),
                          (s 0) + (s s s 0), ...,
(s s 0) + 0, (s s 0) + (s 0), (s s 0) + (s s 0),
                        (s s 0) + (s s s 0), ...,
... ...
0 + (0 + 0), 0 + (0 + (s 0)), ...
...
(0 + 0) + 0, (0 + (s 0)) + 0, ...
...
. }
```

Natural numbers with addition
-- equations define meaning/function

CafeOBJ module PNAT+ defining
Peano Natural numbers with addition

```
mod! PNAT+ {
  pr(PNAT)
  op _+_ : Nat Nat -> Nat .
  vars N1 N2 : Nat .
  -- equations
  eq 0 + N2 = N2 .
  eq (s N1) + N2 = s(N1 + N2) .
}
```

Inference/Computation
with the equations

```
   (s s 0) + (s 0)
= s((s 0) + (s 0))
= s s(0 + (s 0))
= s s s 0
```

```
CafeOBJ> select PNAT+
PNAT+> red s s 0 + s 0 .
PNAT+> -- reduce in PNAT+ :
((s (s 0)) + (s 0)):Nat
(s (s (s 0))):NzNat
(0.000 sec for parse,
3 rewrites(0.000 sec),
5 matches)
```

Reduction of CafeOBJ is honest to equational reasoning

- The **basic mechanism of CafeOBJ verification is equational reasoning. Equational reasoning is to deduce an equation (a candidate of a theorem) from a given set of equations (axioms of a specification).**

- **The CafeOBJ system supports an automatic equational reasoning based on term rewriting.**

- **"`reduce`" or "`red`" command of CafeOBJ helps to do equational reasoning by term rewriting.**

What can be done with `red` (reduction) command?

Let us fix a context M (a module M in CafeOBJ), and let (**`t1 =*M> t2`**) denote that **`t1`** is reduced to **`t2`** in the context. That is, (**`red in M : t1 .`**) returns **`t2`** . Let (**`t1 =M t2`**) denote that **`t1`** is equal to **`t2`** in the context **`M`**. That is (**`t1 = t2`**) can be inferred by equational reasoning in **`M`**. It is important to notice:

(**`t1 =*M> t2`**) implies (**`t1 =M t2`**)

but

(**`t1 =M t2`**) does not implies (**`t1 =*M> t2`**)

Proof Score
for the proof of right zero of addition _+_

```
-- opening module PNAT+
open PNAT+
--> declare that n stands for any Nat value
op n : -> Nat .

**> Proof of (n + 0 = n) by induction on n
**> base case of n = 0:
red 0 + 0 = 0 .
**> induction hypothesis:
eq n + 0 = n .
**> induction step proof for (s n):
red s n + 0 = s n .
**> QED {end of proof}
close
```

Declaring constants and equations then reduce

While a module is opened, declaring constants and equations represents assumptions for equational reasoning done by **red**.

```
%PNAT+> op n : -> Nat .
...
%PNAT+> **> induction hypothesis:
%PNAT+> eq n + 0 = n .
%PNAT+> **> induction step proof for (s n):
**> induction step proof for (s n):
%PNAT+> red s n + 0 = s n .
*
-- reduce in %PNAT+ : (((s n) + 0) = (s n)):Bool
(true):Bool
```

This is a proof of
"**+(*N*, 0) = *N*** implies **+(s(*N*), 0) = s(*N*)** for any natural number ***N***".

Proof Score
for the proof of associativity of addition _+_

```
-- opening module PNAT+
open PNAT+
--> declaring constants for arbitrary values
ops n1 n2 n3 : -> Nat .
**> Proof of associativity:
**>     (n1 + n2) + n3 = n1 +(n2 + n3)
**> by induction on n1
**> base case proof for 0:
red 0 + (n2 + n3) = (0 + n2) + n3 .
**> induction hypothesis:
eq (n1 + n2) + n3 = n1 + (n2 + n3) .
**> induction step proof for (s n1):
red ((s n1) + n2) + n3 = (s n1) + (n2 + n3) .
**> QED {end of proof}
close
```

Comments

A line beginning with “--” (or “”) is ignored, and**
A line beginning with “-->” (or “>”) is echoed back.**

```
CafeOBJ> -- this is a comment
CafeOBJ>
```

```
CafeOBJ> ** this is a comment
CafeOBJ>
```

```
CafeOBJ> --> this is a comment
--> this is a comment
CafeOBJ>
```

```
CafeOBJ> **> this is a comment
**> this is a comment
CafeOBJ>
```

It is very important to write as much appropriate comments as possible for explaining specifications and proof scores (verifications/proofs).

Three kinds of modules

CafeOBJ specification is composed of modules.
There are three kinds of modules.

```
mod! <module_name> {
  <modlue_element> *
}
```

```
mod* <module_name> {
  <modlue_element> *
}
```

```
mod <module_name> {
  <modlue_element>*
}
```

mod! declares that the module denotes tight denotation
mod* declares that the module denotes loose denotation
mod declares that the module does not declare any semantic denotation

[Naming convention] module name starts with two successive upper case characters
(example: **TEST, NAT, PNAT+, ACCOUNT-SYS,**…)

A module is composed of signature and axioms/equations

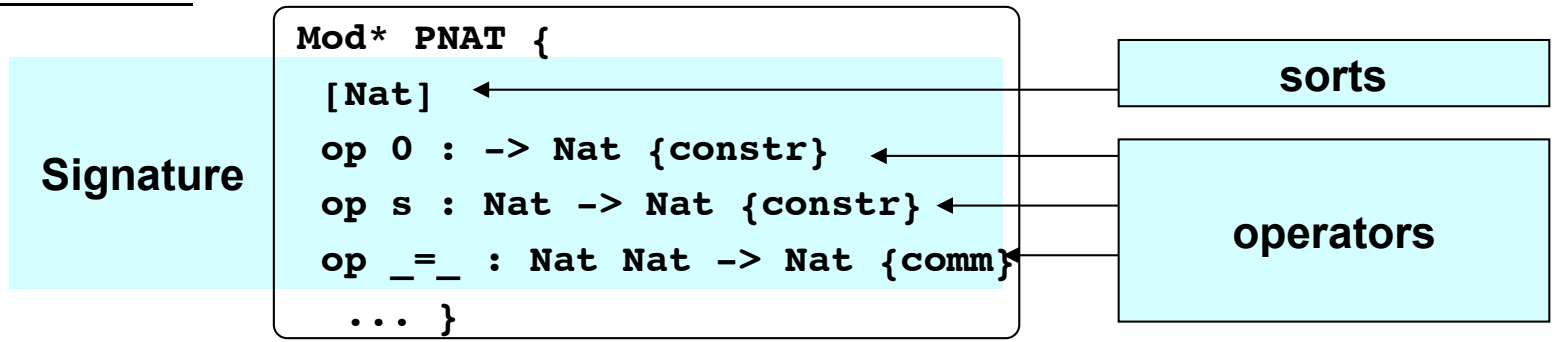
```
mod* PNAT {
  [ Nat ]
  op 0 : -> Nat {constr} .
  op s_ : Nat -> Nat {constr} .
  op _=_ : Nat Nat -> Bool {comm} .

  eq (N:Nat = N) = true .
  eq (0 = s(N2:Nat)) = false .
  eq (s(N1:Nat) = s(N2:Nat))
           = (N1 = N2) .
}
```

signature

axioms/equations

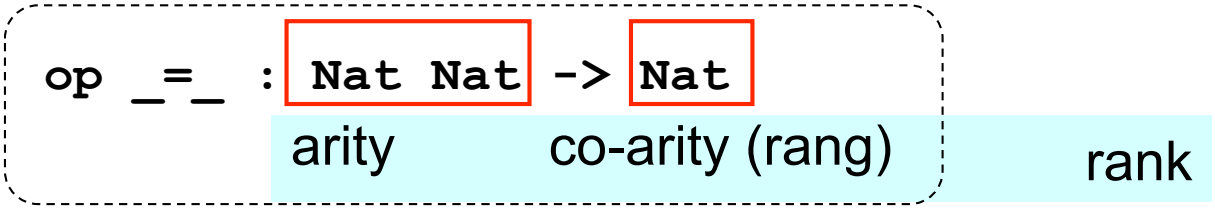
Signature:
sort name, operator name, arity, co-arity, rank

A signature is a pair of a set of sorts and a set of operations.



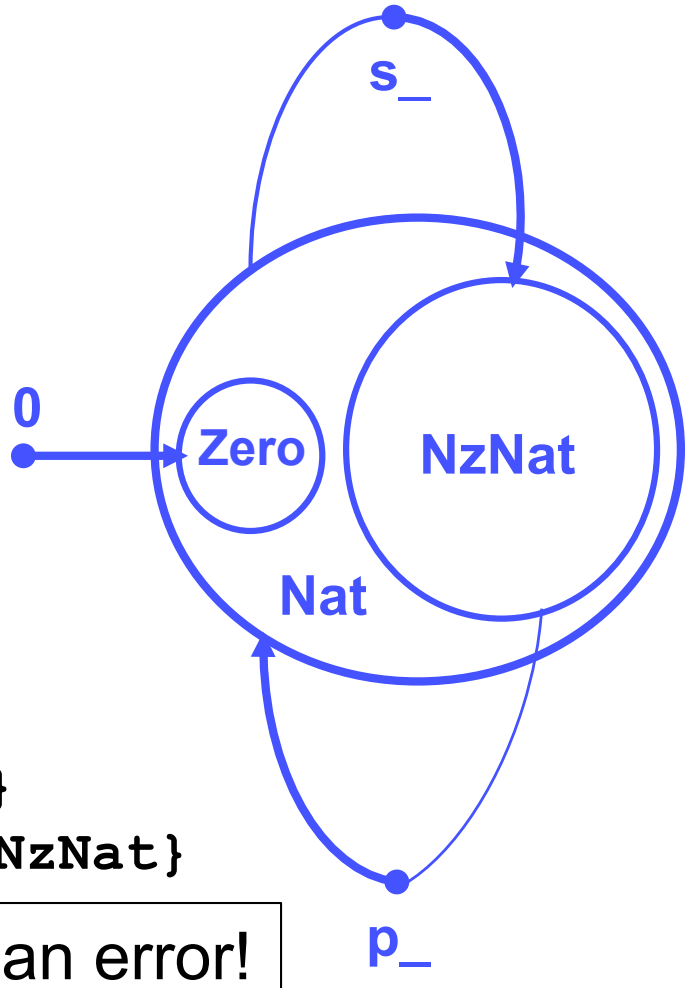
[Convention] The first and second letter of a sort name is written in a upper case and lower case letter respectively. (E.g. `Nat, Set`)

[Convention] The first letter of an operation name is written in a lowerl case letter or a non-alphabet letter. (E.g. `0, s, +`)



Order sorted signature and sorted terms
-- Natural numbers with predecessor function

```
-- signature
-- sorts
[ Zero NzNat < Nat ]
-- operators
op 0 :   -> Zero {constr}
op s_ : Zero -> NzNat {constr}
op s_:  NzNat -> NzNat {constr}
op p_:  NzNat -> Nat
eq p s N:Nat = N .
```



Sorted terms

Zero = {0}

NzNat = {s 0} ∪ {s *n* | *n* ∈ NzNat}

Nat = Zero ∪ NzNat ∪ {p *n* | *n* ∈ NzNat}

`(p 0)` is handled as an error!

Recursive definition of terms
- term is also called expression or tree

For a given signature, `t` is a term of a sort `S` if and only if `t` is

- a variable `X:S`,
- a constant `c` declared by "`op c : -> S`", or
- a term $f(t_1,,t_n)$ for "`op f : S`$_1$ `S`$_n$ `-> S`" and a term t_i of a sort S_i (i =1, ,n).
- a term of a sort `S'` which is a sub-sort of `S` (Example: Since `Zero < Nat`, a term `0:Zero` is also a term of sort `Nat`)

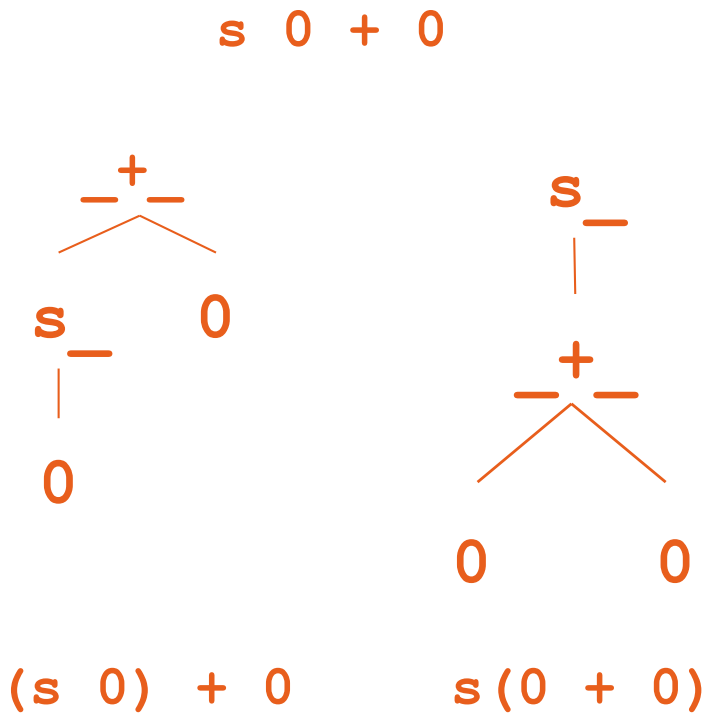
Several forms of function application: standard, prefix, infix, postfix, distfix

```
op f : Nat Nat -> Nat .
    f(2,3)  standard
op (f_ _) : Nat Nat -> Nat .   -- recommended
                               -- for succesive __
    (f 2 3) prefix
op f__ : Nat Nat -> Nat .
    (f 2 3) prefix
op _+_ : Nat Nat -> Nat .
    (2 + 3) infix
op  _! : Nat -> Nat .
    (5 !)   postfix
op if_then_else_fi : Bool Nat Nat -> Nat .
    (if 2 < 3 then 4 else 5 fi)  distfix
```

“(“ and “)” are meta-charactors for grouping expressions in CafeOBJ and can not be used foranother purpose.

Parsing – precedence of operators

`s 0 + 0` represents `(s 0) + 0`, because the operator `(s _)` has high precedence than the operator `(_ + _)`



The preceedences of the operators can be checked by the commands

```
describe op (s _)
describe op (_ + _)
```

Equation

An equation is a pair of terms of a same sort, and written as:

```
eq l = r .
```

in CafeOBJ. Where `l` is called the left-hand side (LHS) of the equation and `r` is the right-hand side (RHS). An equation can have a condition (COND) `c` like:

```
ceq l = r if c .
```

- **Most important kind of axioms of CafeOBJ specification are equations**
- **Properties to be verified are also expressed as equations**

Conditions for an equation to be a rewriting rule

For an equation to be used as a rewriting rule for doing reductions, the following conditions must be satisfied.

(1) LHS is not a variable.

an example violating this condition:

```
eq N:Nat = N:Nat + 0 .
```

(2) All variables in RHS are in LHS.

an example violating this condition:

```
eq 0 = N:Nat * 0 .
```

Two way of declaring variables
- use appropriate one based on the situation

Variable can be declared in an equation directly. The scope of the variable ends at the end of the equation.

```
mod! PNAT+ { [Nat] …
  eq      0       + N2:Nat = N .
  eq (s N1:Nat) + N2:Nat = s(N1 + N2) .
}
```

Variables can be declared before equations. The scope of the variable ends at the end of the module.

```
mod! PNAT+ { [Nat] …
  vars M N : Nat .
  eq    0  + N = N .
  eq (s M) + N = s(M + N) .
}
```

Two equality predicates _=_ and _==_

Assume that (**t1 =*> t1'**) and (**t2 =*> t2'**) in any context
then
if (**t1'** and **t2'** are the same term)
then (**red t1 = t2 .**) returns **true**
and
(**red t1 == t2 .**) returns **true**
if (t1' and t2' are different terms)
then (**red t1 = t2 .**) returns (**t1' = t2'**)
but
(**red t1 == t2 .**) returns **false**

If reduction/rewriting is not complete w.r.t. a set of equations, _==_ may returns false even if two terms may have a possibility of being equal w.r.t. the set of equations.

Constant v.s. variable

Using a variable in an equation instead of a constant makes a drastic change of meaning of the proof score. Be careful!

- **The scope of a constant is to the end of a open-close session assuming that the declared constants are fresh.**
- **The scope of a variable is inside of the equation.**

```
open PNAT+
op n : -> Nat .
eq n + 0 = n .
red (s n) + 0 = s n .
close
```

```
open PNAT+
var N : Nat .
Eq N + 0 = N .
red (s N) + 0 = s N .
close
```

Constant: $\forall N{:}Nat.\ [+(N,0)=N \Rightarrow +(s(N),0)=s(N)]$

Variable: $\forall N{:}Nat.[+(N,0)=N\ ] \Rightarrow \forall N{:}Nat.[\ +(s(N),0)=s(N)]$

Exercise

```
mod! PNAT+* { pr(PNAT)
  vars X Y Z : Nat .
  op _+_ : Nat Nat -> Nat {prec: 30}
  eq 0 + Y = Y .
  eq s(X) + Y = s(X + Y) .
  op _*_ : Nat Nat -> Nat {prec: 29}
  eq 0 * Y = 0 .
  eq s(X) * Y = Y + (X * Y) .   }
```

Write proof scores to verify that binary operators _+_ and _*_ in PNAT+* are associative and commutative. Write also proof scores to verify that _*_ distributes over _+_, that is

(N1 + N2) * N3 = (N1 * N3) + (N2 * N3) .