Basics of CafeOBJ and Peano Style Natural Numbers

Lecture Note 1

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

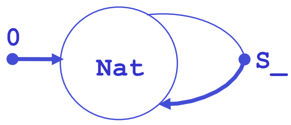
Natural Numbers -- Signature --

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` :	if 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

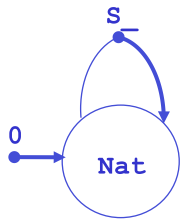
```
mod! PNAT {
  [ Nat ]
  op 0 : -> Nat {constr} .
  op s_ : Nat -> Nat {constr} .

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

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

Natural numbers
-- signature and expressions/terms

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



Nat = { 0 } ∪ { s n | n ∈ Nat }

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.

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

<u>Induction Scheme:</u>

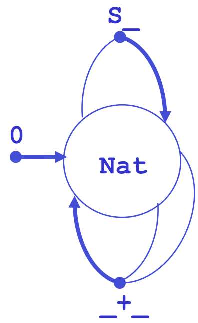
$$\frac{P(0) \qquad \forall n \in N.[P(n) \Rightarrow P(s\ n)]}{\forall n \in N.P(n)}$$

<u>Concrete Procedure:</u> (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
   -- _+_ is a defined operator
```



$\text{Nat} = \{ 0 \} \cup \{ \text{s}\ n \mid n \in \text{Nat} \}$

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

Natural numbers with addition
-- expressions/terms composed by operators

```
NatExp = {
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, ...
...
. }
```

Because _+_ is a defined operator, any _+_ operator is supposed to be eliminated. That is, NatExp ==> Nat .

Natural numbers with addition
-- equations define meaning/function

CafeOBJ module PNAT+ defining Peano Natural numbers with addition

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

Defined operator (_+_) is erased by the two equations.

Sufficient Completeness

Computation/inference 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))):Nat
(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 right zero property: (N:Nat + 0 = N)

```
-- proof by induction with respect to N:Nat
-- induction base case:
-- opening module PNAT+ to make use of all its contents
open PNAT+
red 0 + 0 = 0 .
close
-- induction step case:
open PNAT+
-- declare that the constant n stands for any Nat value
op n : -> Nat .
-- induction hypothesis:
eq n + 0 = n .
-- induction step proof for (s n):
red s n + 0 = s n .
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
$\forall$ ***N*:Nat.** [**(*N* + 0) = *N*** implies (**(s *N*) + 0) = (s *N*)**].

Proof score for associativity of (_ + _) (N1:Nat + N2:Nat) + N3:Nat = N1 +(N2 + N3)

```
**> induction base case:
open PNAT+
red 0 + (N2:Nat + N3:Nat) = (0 + N2) + N3 .
Close
**> induction step case:
open PNAT+
**> declare that the constant n1 stands for any Nat value
op n1 : -> Nat .
**> induction hypothesis:
eq (n1 + N2:Nat) + N3:Nat = n1 + (N2 + N3) .
**> induction step proof for (s n1):
red ((s n1) + N2:Nat) + N3:Nat = (s n1) + (N2 + N3) .
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 **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 .
  ceq (N1 = N2) if (N1 = N2) .
  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 <u>sorts</u> and a set of <u>operations</u>.

Signature

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

sorts

operators

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

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

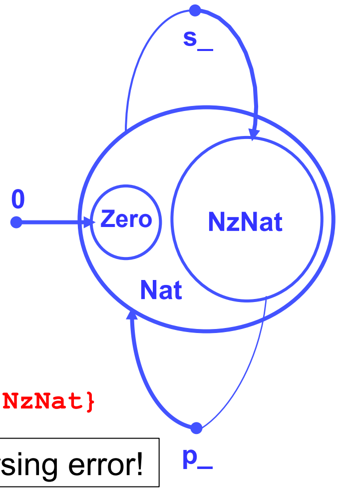
`op _=_ : Nat Nat -> Bool`

arity co-arity

rank

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

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



Sorted terms

```
Zero = {0}
NzNat = {s n | n ∈ Nat}
Nat = Zero ∪ NzNat ∪ {p n | n ∈ NzNat}
```

`(p 0)` is handled as a parsing 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 of sort Zero` is also a term of sort `Nat`)

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

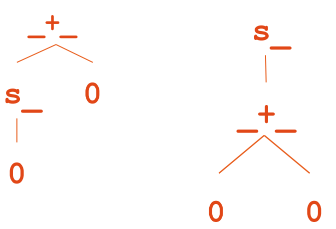
```
op f : Nat Nat -> Nat .
    f(2,3)  standard
op (f_ _) : Nat Nat -> Nat .   -- recommended
                               -- for succesive __
    (f 2 3) prefix
op f__ : Nat Nat -> Nat .
    (f 2 3) prefix
op _+_ : Nat Nat -> Nat .
    (2 + 3) infix
op  _! : Nat -> Nat .
    (5 !)   postfix
op if_then_else_fi : 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 for any other purpose.

Parsing – precedence of operators

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

`s 0 + 0`



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

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, a Boolean term) `c` like:

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

- **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 = N2 .
  eq (s N1:Nat) + N2:Nat = s(N1 + N2) . }
```

Variables can be declared before equations. This is just abbreviation for saving many variable declarations in the equations. N2 in the first eq has nothing to do with N2 in the second eq .

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


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

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

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.

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) .

Module PNAT+

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

  eq[+1]: 0 + Y:Nat = Y .
  eq[+2]: (s X:Nat) + Y:Nat = s(X + Y) .
  eq[m2o]: (X:Nat = X) = true .
  ceq[o2m]: X:Nat = Y:Nat if X = Y .
  eq (0 = s Y:Nat) = false .
  eq (s X:Nat = s Y:Nat) = (X = Y) .
}
```

Signature Σ

Equations E

Goals or entailments

A primary goal for writing spec S = <Σ,E> is to prove that any model of S satisfy a (conditional) equation $(\forall \mathbf{X})e$. The goal is written as:

$$E \models (\forall \mathbf{X})e \quad \text{or} \quad S \models (\forall \mathbf{X})e \,.$$

These goals are sometimes called semantic entailments.

For doing formal verification, it is common to think of entailment:

$$E \vdash (\forall \mathbf{X})e$$

which corresponds to semantic entailment:

$$E \models (\forall \mathbf{X})e \,.$$

We have a *quasi* complete set of proof rules for entailments which satisfies:

$$E \vdash (\forall \mathbf{X})e \quad \text{iff} \quad E \models (\forall \mathbf{X})e$$

Examples of Goals

PNAT+ |- (∀X:Nat)(X + 0 = X)
PNAT+ |- (∀X:Nat)(∀Y:Nat)(X + (s Y) = s (X + Y))
PNAT+ |- (∀X:Nat)(∀Y:Nat)(X + Y = Y + X)
in standard notation

PNAT+ |- (X:Nat + 0 = X)
PNAT+ |- (X:Nat + (s Y:Nat) = s (X + Y))
PNAT+ |- (X:Nat + Y:Nat = Y + X)
in CafeOBJ notation

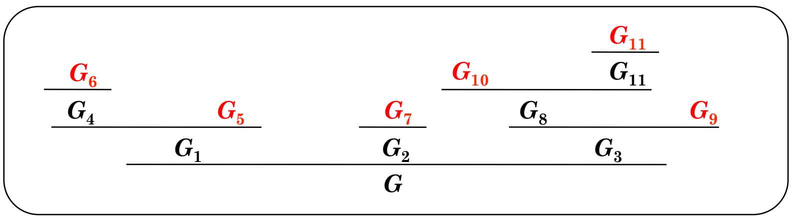
Proof Tree of Goal

A proof of a goal *G* is a tree of goals (called a *proof tree* of G) such that

- ***G* is the root,**
- **for each node *N*, its sub-nodes $SN_1,\ldots,SN_m$ are generated by applying a (derived) proof rule to the node. That is, by applying the following (derived) proof rule:**

$$\frac{SN_1,\ldots,SN_m}{N}$$

- **and, each leaf can be discharged by CafeOBJ reduction/rewriting.**



Proof Tree of (∀X)(X + 0 = X)

$$\dfrac{\text{PNAT+} \vdash (0 + 0 = 0) \qquad \text{PNAT+} \cup \{x + 0 = x\} \vdash_{\{x\}} (s\ x) + 0 = s\ x}{\text{PNAT} \vdash (\forall X)(X + 0 = X)}\ ①$$

(① Struct Ind)

✓ Each leaf can be discharged by rewriting.

- $0 + 0 \rightarrow 0$ by [+1]
- $(s\ x) + 0 \rightarrow s\ (0 + x)$ by [+2]
  $\rightarrow s\ x$ by I.H.

Proof Tree of (∀X)(∀Y)(X + s Y = s (X + Y))

$$\dfrac{\dfrac{\text{PNAT} \vdash_{\{x\}} 0 + s\ y = s\ (0 + y)}{\text{PNAT+} \vdash (\forall Y)\ (0 + s\ Y = s\ (0 + Y))}\ ② \qquad \dfrac{\text{PNAT+} \cup \{(\forall Y)(x + s\ Y = s\ (x + Y))\} \vdash_{\{x,y\}} s\ x + s\ y = s(s\ x + y)}{\text{PNAT+} \cup \{(\forall Y)(x + s\ Y = s(x + Y))\} \vdash_{\{x\}} (\forall Y)(s\ x + s\ Y = s(s\ X + Y))}\ ③}{\text{PNAT+} \vdash (\forall X)(\forall Y)(X + s\ Y = s\ (X + Y))}\ ①$$

(① Struct Ind) (② Generalization) (③ Generalization)

✓ Each leaf can be discharged by rewriting.

Discharge of Goals with CafeOBJ codes

CafeOBJ can check if goals are discharged.

PNAT+∪{x + 0 = x} $\vdash_{\{x\}}$ (s x) + 0 = s x

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

PNAT+∪{(∀Y)(x + s Y = s (x + Y))} $\vdash_{\{x,y\}}$ s x + s y = s(s x + y)

```
open PNAT+
 ops x y : -> Nat .
 eq x + s Y:Nat = x + Y .
 red s x + s y = s(s x + y) .
close
```


Proof Passages & Scores

- A CafeOBJ code fragment that checks if a goal is discharged is called a *proof passage* of the goal.
- The set of the proof passages of the leaves of a proof tree of a goal is called a *proof score* of the goal.

A proof score of PNAT+ |– (∀X)(∀Y)(X + Y = Y + X)

```
open PNAT+
 op y : -> Nat .
 eq 0 + X:Nat = X .
 red 0 + y = y + 0 .
close
```

A proof passage of
PNAT+∪{(∀X)(X + 0 = X)} $\vdash_{\{y\}}$ 0 + y = y + 0

A proof passage of
PNAT+∪{(∀Y)(x + Y = Y + x),(∀X)(∀Y)(X + s Y = s (X + Y)} $\vdash_{\{x,y\}}$ s x + y = y + s x

```
open PNAT+
 ops x y : -> Nat .
 eq x + Y:Nat = Y + x .
 eq X:Nat + s Y:Nat = s(X + Y) .
 red s x + y = y + s x .
close
```