# Parameterized Modules and Generic List Structure

**FUTATUSGI, Kokichi**
二木 厚吉
**JAIST**

# Topics

- Parameterized module and generic list
- Views, on the fly view decl., module expressions
- Induction over List
- Verifications of generic list by proof scores

# Parameterized Module `LIST`

```
mod* TRIV= {
   [Elt]
   op _=_ : Elt Elt -> Bool {comm} .
   }

mod! LIST (X :: TRIV=) {
   [List]
   op nil : -> List {constr} .
   op _|_ : Elt.X List -> List {constr} .
     ...  }
-- Elt.X indicates Elt in (X :: TRIV=)
```

# view from TRIV= to PNAT

```
mod* TRIV= { [Elt]
             ... }
```

formal param.

actual param.

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

```
view TRIV=>PNAT= from TRIV= to PNAT {
  sort Elt -> Nat,
  op (E1:Elt = E2:Elt) -> (E1:Nat = E2:Nat) }
```

# Instantiation of parameterized module with view and renaming

```
mod PNAT=LIST
{pr(
      LIST(TRIV=>PNAT=)
      *{sort List -> NatList}
   )
}
```

```
make PNAT=LIST
(
 LIST(TRIV=>PNAT=)
 *{sort List -> NatList}
)
```

# View calculus (or view inference)

The following three views are defining the same view.

```
view TRIV=>PNAT= from TRIV= to PNAT {
  sort Elt -> Nat,
  op _=_ -> _=_ }

view TRIV=>PNAT= from TRIV= to PNAT {
  sort Elt -> Nat }

view TRIV=>PNAT= from TRIV= to PNAT { }
```

View is calculated using,
(1) Sort and operator mapping information given by view,
(2) Principal sort correspondence,
(3) Equality of sort name and operator name,
(4) Induced condition from sort map on rank of a target operator.

# On the fly view definition in instantiation

```
make PNAT=LIST
(LIST(X <= view to PNAT
          {sort Elt -> Nat, op _=_ -> _=_})
   *{sort List -> NatList})

--> another way to define PNAT=LIST
make PNAT=LIST
(LIST(PNAT{sort Elt -> Nat, op _=_ -> _=_})
   *{sort List -> NatList})

--> yet another way to define PNAT=LIST
make PNAT=LIST
(LIST(PNAT)*{sort List -> NatList})
```

# Target of an operator can be a term (derived op) in view definition

```
make NAT<=>LIST
 (LIST(NAT{sort Elt -> Nat,
           op (E1:Elt = E2:Elt) ->
              ((E1:Nat <= E2:Nat) and
               (E1:Nat >= E2:Nat))}))
```

NAT is built-in module of natural numbers. The module NAT contains (1) sort Nat which is a set of infinite natural numbers, and (2) ordinary fundamental operations over Nat.

# Module expression

**A module expression is an expression composed of followings five kinds of components.**

**(1) module names**
**(2) parameterized module names**
**(3) view names and on-the-fly view definitions**
**(4) renamings**
**(5) module sums (e.g. ME1 + ME2)**

1 The same module expressions which appear as arguments of (_ + _) shrink into one.

2 Two same module sub-expressions (except module names) which appear in a module expression create two different modules.

# An example of module expression

```
(PAIR(LIST(PNAT){sort Elt -> List},
      LIST(PNAT){sort Elt -> List})
 +
 LIST(PNAT)*{sort List -> NatList}
 +
 LIST(PNAT)*{sort List -> NatList}
 +
 LIST(STRING
      {sort Elt -> String,
       op (E1:Elt = E2:Elt)
          -> string=(E1:String,E2:String)})
  *{sort List -> StringList})
```

1. The first and the second LIST(PNAT{…}) create different modules.
   (This creates serious errors and should be avoided!)
2. The third and fourth LIST(PANT)*{…} shrink into one.

# Three modes of module importation

Semantics definition of three modes

protecting (pr) : no junk and no confusion into the imported module

extending (ex) : may be junk but no confusion into the imported module

**including (inc) : no semantic declaration for imported module, but make sub-module structure**

No semantics checks are done by CafeOBJ system w.r.t. protecting and extending

```
mod 2PNATlist{inc(LIST(PNAT)) inc(LIST(PNAT)}
```

creates two different modules with the same name. (avoid it!)

# Order-sorted parameterized list and error handling

```
mod! LISTord (X :: TRIV=)
principal-sort List
{[NnList < List]
 op nil : -> List {constr} .
 op _|_ : Elt.X List -> NnList {constr} .
 -- taking head of list
 op hd_ : NnList -> Elt.X .
 eq hd (E1:Elt.X | L1:List) = E1 .
 -- taking tail of list
 op tl_ : NnList -> List .
 eq tl (E1:Elt.X | L1:List) = L1 .
 ... }
```

**The followings are error terms:**
**(hd nil):?Elt  (tl nil):?List**

# Inductive/Recursive definition of the sort `List` generated by constructors

```
Elt =  {e1, e2, e3, …}
```

```
[NnList < List]
op nil : -> List {constr}
op _|_ : Elt List -> NnList {constr}
```

```
NnList = {(e | l)| e∈Elt, l∈List}
List = {nil}∪ NnList
```

```
[List]
op nil : -> List {constr}
op _|_ : Elt List -> List {constr}
```

```
List = {nil} ∪ {(e | l)| e∈Elt, l∈List}
```

# Mathematical Induction over generic list -- induced by declaration of constructors

**The inductive structure defined by two constructors of sort List induces the following induction scheme.**

Goal: Prove that a property $P(l)$ is true for any list $l$ .

Induction Scheme:

$$\frac{P(\text{nil}) \qquad \forall L{:}\text{List}.[P(L) \Rightarrow \forall E{:}\text{Elt}.P(E \mid L)]}{\forall L{:}\text{List}.P(L)}$$

Concrete Procedure: (induction with respect to l)

1. Prove $P(\text{nil})$ is true
2. Assume that $P(l)$ holds,
   and prove that $P(e \mid l)$ is true

# Append _@_ over List

```
--> append _@_ over List
mod! LIST@(X :: TRIV=) {
  pr(LIST(X))
  -- append operation over List
  op _@_ : List List -> List .
  var E : Elt .
  vars L1 L2 : List .
  eq[@1]: nil @ L2 = L2 .
  eq[@2]: (E | L1) @ L2 = E | (L1 @ L2) .
}
```

An axiom can be named by putting a name like "`[@1]:`".

# Trace command

```
%LIST@(X)> set trace whole on
red (a | b | c | nil) @ (a | b | c | nil) .
set trace whole off
%LIST@(X)> -- reduce in %LIST@(X) : ...
[1]: ((a | (b | (c | nil))) @ (a | (b | (c | nil))))
---> (a | ((b | (c | nil)) @ (a | (b | (c | nil)))))
[2]: (a | ((b | (c | nil)) @ (a | (b | (c | nil)))))
---> (a | (b | ((c | nil) @ (a | (b | (c | nil))))))
[3]: (a | (b | ((c | nil) @ (a | (b | (c | nil))))))
---> (a | (b | (c | (nil @ (a | (b | (c | nil)))))))
[4]: (a | (b | (c | (nil @ (a | (b | (c | nil)))))))
---> (a | (b | (c | (a | (b | (c | nil))))))
(a | (b | (c | (a | (b | (c | nil))))))):List
(0.000 sec for parse, 4 rewrites(0.000 sec), 7 matches)
```

# nil is right-identity of _@_ (@ri)

```
-->     eq[@ri]: L:List @ nil = L .
--> ========================================
-- Proof: By induction on L:List
--> I Induction Base
select LIST@ .
red (nil @ nil) = nil .
--> II Induction Step
open LIST@ .
-- induction hypothesis
op l : -> List . eq (l @ nil) = l .
-- check
red (`e:Elt | l) @ nil = (`e | l) .
close
--> QED
```

# _@_ is associative (@assoc)

```
--> eq[@assoc]:
-->   (L1:List @ L2:List) @ L3:List= L1 @ (L2 @ L3) .
--> ==========================================
-- Proof: By induction on L1
--> I Induction Base
select LIST@ .
red (nil @ `l2:List) @ `l3:List = nil @ (`l2 @ `l3) .
--> II Induction Step
open LIST@ .
-- induction hypothesis,
op l1 : -> List .
eq (l1 @ L2:List) @ L3:List = l1 @ (L2 @ L3) .
-- check
red ((`e:Elt | l1) @ `l2:List) @ `l3:List
    = (`e:Elt | l1) @ (`l2 @ `l3) .
close
--> QED
```

# Reverse operations on lists

```
mod! LISTrev(X :: TRIV=) {
  pr(LIST@a(X))
  vars L L1 L2 : List .
  var E : Elt.X .
  -- one argument reverse operation
  op rev1 : List -> List .
  eq rev1(nil) = nil .
  eq rev1(E | L) = rev1(L) @ (E | nil) .
  -- two arguments reverse operation
  op rev2 : List -> List .
  -- auxiliary function for rev2
  op sr2 : List List -> List .
  eq rev2(L) = sr2(L,nil) .
  eq sr2(nil,L2) = L2 .
  eq sr2(E | L1,L2) = sr2(L1,E | L2) .
}
```

# Exercises

With respect to the module LISTrev, write proof scores for verifying the followings.

```
(1) (∀L:List).(rev1(rev1(L) = L))
    eq rev1(rev1(L:List)) = L .
(2) (∀L:List).(rev1(L) = rev2(L))
    eq rev1(L:List) = rev2(L) .
```