Parameterized Modules and Generic List Structure

Lecture Note 2

Topics

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

Starting System

CafeOBJ system is invoked by typing "cafeobj", and the system waits for your input with a "prompt".

```
bash-3.2$ cafeobj
-- loading standard prelude
; Loading /usr/local/cafeobj-1.4/prelude/std.bin

          -- CafeOBJ system Version 1.4.8(PigNose0.99,p10) --
                   built: 2009 Aug 10 Mon 6:49:29 GMT
                         prelude file: std.bin
                                  ***
                      2009 Dec 3 Thu 11:40:41 GMT
                            Type ? for help
                                  ***
                  -- Containing PigNose Extensions --
                                  ---
         built on International Allegro CL Enterprise Edition
              8.1 [Mac OS X (Intel)] (Aug 10, 2009 15:49)
processing input : /Users/kokichi/.cafeobj
CafeOBJ>
```

.cafeobj file is loaded

"Prompt" from CafeOBJ system shows the current module

Quitting System

By typing "quite" or "q" (or typing control-D; this depends on the OS you are using), you can quite from the CafeOBJ system

```
CafeOBJ> quit
[Leaving CafeOBJ]
%
```

```
CafeOBJ> q
[Leaving CafeOBJ]
%
```

Inputting modules or selecting modules

In the top level of CafeOBJ system, the commands for inputting modules and selecting modules are available.

```
CafeOBJ> mod TEST { [ Elt ] }
 -- defining module TEST_* done.
CafeOBJ>
```

input

output

After a inputting module, the module is available by typing "selecting <the module name>"

```
CafeOBJ> select TEST

TEST>
```

After selecting the module "ModName", the prompt is changed to "ModName>" .

Inputting files

It is always recommended that CafeOBJ codes is prepared in some file and the file is inputted into CafeOBJ system by typing "in <fileName>" or "input <fileName>". The file extension of CafeOBJ file is ".cafe" .

example: inputting "test.cafe" file

```
% more test.cafe
mod TEST { [Elt] }
select TEST
%
```

contents of the file "test.cafe"

```
CafeOBJ> in test.cafe
processing input : /…/test.cafe
-- defining module TEST._* done.
TEST>
```

File name extension of ".cafe" will be available shortly!

System Error -- error should be eliminated -- warning needs not be eliminated, but recommended to be eliminated

CafeOBJ system report an error as follows:

```
CafeOBJ> mod ERROR }
[Error]: was expecting the symbol `{' not `}' .
CafeOBJ>
```

"[Error]…" reports a serious error like syntax error in inputted CafeOBJ code. CafeOBJ system may go down to CHAOS level for some special errors.

```
CafeOBJ> ^C
Error: Received signal number 2 …
[1c] CHAOS(1):
```

From CHAOS level, CafeOBJ system will be recovered by typing ":q" in almost all cases. ^C (control C) make you get into CHAOS levlel.

```
[1c] CHAOS(1): :q
CafeOBJ>
```


?, show, describe, show ? commands

- By typing "?", you can see the list of commands which are available at the level.
- "show" command shows varieties of information
  - "show <module name>", "show sorts", "show ops"
  - "show ?" gives you a list of show commands available
- "show" can be shortened into "sh"

```
CafeOBJ> ?
-- CafeOBJ top level commands :
-- Top level definitional forms include `module'(object, theory),
-- `view', and `make'
  ?                  print out this help
  quit -or-
  q                  exit from CafeOBJ interpreter
  select <Modexp>    set the <Modexp> current
  show -or-
  describe           print various info., for further help, type `show ?'
  ...
```

set, set ? and show switches command

set command set switches of CafeOBJ system. By changing switches you can customize CafeOBJ system get different behaviors of the system.

```
CafeOBJ> set auto context on
CafeOBJ> mod TEST { [ Elt] }
-- defining module TEST._* done.
TEST>
```

making system select the last entered module automatically

```
TEST> set ?
TEST> ...
```

Shows all `set` commands

```
TEST> show switches
TEST> ...
```

Shows all switches

Parameterized Module LIST

```
mod* TRIV= {
  [Elt]
  op _=_ : Elt Elt -> Bool {comm} .
  eq (E:Elt = E) = true .
  ceq E1:Elt = E2:Elt if (E1:Elt = E2:Elt) . }

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

view from TRIV= to PNAT

```
mod* TRIV= { [Elt]
  op _=_ : Elt Elt -> Bool {comm} .
  eq (E:Elt = E) = true . }
```

formal param.

→

```
--> Peano style natural numbers
mod! PNAT { [ Nat ]
  op 0 : -> Nat {constr}
  op s_ : Nat -> Nat {constr}
  op _=_ : Nat Nat -> Bool {comm}
  eq (X:Nat = X) = true . ... }
```

actual param.

```
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 is 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 _=_ -> 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)*{…} shrinks 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 modules with the same name. (avoid it!)

Order-sorted parameterized list and error handling

```
mod! LISTord (X :: TRIV=) {
  [Nil NnList < List]
  op nil : -> Nil {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, …}
```

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

```
Nil = {nil}
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:

    P(nil)    ∀L:List.[P(L) => ∀E:Elt.P(E | L)]
  ------------------------------------------------
                  ∀L:List.P(L)

Concrete Procedure: (induction with respect to l)
      1. Prove P(nil) is true
      2. Assume that P(l) holds,
         and prove that P(e | 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)
```


Definitions of properties about _@_

```
--> properties about _@_
mod PROP-LIST@(X :: TRIV=) {
  pr(LIST@(X))
  -- CafeOBJ variables
  vars L1 L2 L3 : List .
  -- nil is right identity of _@_
  op @ri : List -> Bool .
  eq @ri(L1) = ((L1 @ nil) = L1) .
  -- _@_ is associative
  pred @assoc : List List List .
  eq @assoc(L1,L2,L3)
     = ((L1 @ L2) @ L3 = L1 @ (L2 @ L3)) .
}
```

Proof score for `@ri(L:Nat)`

```
--induction base
open PROP-LIST@
-- check
  red @ri(nil) .
close
```

@ri(nil)

```
-- induction step
open PROP-LIST@
-- arbitrary values
  op e : -> Elt.X .
  op l : -> List .
-- check
  red @ri(l) implies @ri(e | l) .
close
```

∀L:List.[@ri(L) => ∀E:Elt.@ri(E | L)]

Proof score for `@assoc(L1,L2,L3)`

```
--induction base
open PRED-LIST@
 ops l2 l3 : -> List .
 red @assoc(nil,l2,l3) .
close
```

**∀L1,L2:List.
@assoc(nil,L2,L3)**

```
-- induction step
open PRED-LIST@
 op e : -> Elt.X .
 ops l1 l2 l3 : -> List .
 red @assoc(l1,l2,l3) implies @assoc(e | l1,l2,l3) .
close
```

**∀L1,L2,L3:List.
[@assoc(L1,L2,L3)
=> ∀E:Elt.@assoc(E | L1,L2,L3)]**

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))

(2) (∀L:List).(rev1(L) = rev2(L))

Proof Tree of $(\forall L_1,L_2,L_3)$ sr2@(L_1,L_2,L_3)

Let sr2@(L_1,L_2,L_3) be (sr2(L_1,L_2) @ L_3 = sr2$(L_1,L_2$ @ $L_3)$) .

LISTrev $\vdash_{\{l2,l3\}}$ sr2@(nil,l_2,l_3)

———————————————— ②

LISTrev $\vdash$ $(\forall L_2,L_3)$ sr2@(nil,L_2,L_3)

LISTrev ∪ {$(\forall L_2,L_3)$ sr2@(l,L_2,L_3)}
$\vdash_{\{l,e,l2,l3\}}$ sr2@(e | l,l_2,l_3)

———————————————— ③

LISTrev ∪ {$(\forall L_2,L_3)$ sr2@(l,L_2,L_3)}
$\vdash_{\{l,e\}}$ $(\forall L_2,L_3)$ sr2@(e | l,L_2,L_3)

———————————————— ①

LIST@ $\vdash$ $(\forall L_1,L_2,L_3)$ sr2@(L_1,L_2,L_3)

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

✓ Each leaf can be discharged by rewriting.

Proof Tree of $(\forall L)$(rev1(L) = rev2(L))

LISTrev
$\vdash$ (rev1(nil) = rev2(nil))

LISTrev ∪ {rev1(l) = rev2(l),
$(\forall L_1,L_2,L_3)$(sr2(L_1,L_2 @ L_3)
= rev2(L_1,L_2) @ L_3)}
$\vdash_{\{l,e\}}$ rev1(e | l) = rev2(e | l)

———————————————— ②

LISTrev ∪ {rev1(l) = rev2(l)}
$\vdash_{\{l,e\}}$ rev1(e | l) = rev2(e | l)

———————————————— ①

LISTrev $\vdash$ $(\forall L)$(rev1(L) = rev2(L))

(① Struct Ind) (② Lemma)

✓ Each leaf can be discharged by rewriting.