Programare Logică

CafeObj

http://www.ldl.jaist.ac.jp/cafeobj/

CafeOBJ

- Este dezvoltat la JAIST:
  Japan Advanced Institute of
  Science and Technology

  http://www.ldl.jaist.ac.jp/cafeobj/

- Aparţine familiei OBJ, al cărei părinte a fost J. Goguen(1941 - 2006). Din aceeasi familie de limbaje face parte limbajul Maude, dezvoltat in prezent la University of Illinois at Urbana-Champaign (UIUC) şi Stanford Research Institute (SRI).

  http://maude.cs.uiuc.edu/

CafeOBJ

- Poate fi un instrument util in dezvoltarea de software, deoarece permite atât specificarea, cât şi analiza unui limbaj de programare. Faptul ca este executabil oferă şi avantajul unei implementări indirecte.
- Poate fi folosit ca demonstrator. Mecanismul de rescriere este un procedeu de demonstrare automata, dar sunt implementate si facilităţi speciale.

Sinaia School of Formal Verification of Software Systems

http://www.jaist.ac.jp/ kokichi/class/SinaiaSchoolFVSS0803/

CafeObj

- CafeObj-ul este un interpretor.
- Comenzile sunt introduse una câte una şi sunt executate imediat.
- Un "program" este o mulţime de module.
- Modulele pot fi scrise în fişiere sau direct în linia de comandă.
- Există câteva module predefinite

http://www.jaist.ac.jp/ t-seino/lectures/cafeobj-intro/en/builtin-modules.html

Comenzi

input *cale\nume-fisier.mod*
show *nume-modul* .
select *nume-modul* .
set trace on . / set trace off .
set whole trace on . / set whole trace off .
reduce *termen* .
reduce *termen1* == *termen2* .
parse *term* .
open *nume-modul* .
close
quit

Comenzi

```
CafeOBJ> input proglog\natstr.mod
processing input : proglog\natstr.mod
-- defining module! MYNAT
..................
CafeOBJ> show MYNAT .
module! MYNAT
{
  imports {
    protecting (BOOL)
  }
  signature {
    [ Nat ]
    op 0 : -> Nat
    op s _ : Nat -> Nat
  }
```

Comenzi

```
CafeOBJ> open MYNAT .
-- opening module MYNAT.. done.
%MYNAT> reduce s s s 0 .
-- reduce in %MYNAT : (s (s (s 0))):Nat
(s (s (s 0))):Nat
(0.000 sec for parse, 0 rewrites(0.000 sec), 0 matches)
%MYNAT> parse s s s 0 .
(s (s (s 0))):Nat
%MYNAT> close
CafeOBJ>
```

Modulul predefinit BOOL este importat de orice modul.

Exemplul 1

```
mod! MYNAT{
[Nat]
op 0 : -> Nat
op s_ : Nat -> Nat }
```

Acest modul introduce tipul de date Nat.

Datele de tip Nat sunt:

`0, s 0, s s 0, s s s 0, s s s s 0, s s s s s  0, ...`

Este uşor de văzut că acest modul defineşte numerele naturale.

Modelul matematic:

$S = \{Nat\},\ \Sigma = \{0 :\rightarrow Nat, s : Nat \rightarrow Nat\}$

(S, Σ) este o signatură şi defineşte o clasă de algebre (structuri, structuri algebrice). Algebra $(\mathbb{N}, 0, succesor)$ este un obiect privilegiat al acestei clase (detalii la curs).

Exemplul 2

```
mod! MYNATSTR{ protecting (MYNAT)
[ Nat < NatStr ]
op nil : -> NatStr
op _;_ : NatStr NatStr -> NatStr { assoc }
var L : NatStr
eq nil ;  L = L .
eq L ; nil = L . }
```

Structura generală a unui modul:

- importuri,
- declararea sorturilor si a subsorturilor,
- declararea operaţiilor,
- declararea variabilelor,
- ecuaţii.

Exemplul 2

```
mod! MYNATSTR{ protecting (MYNAT)
[ Nat < NatStr ]
op nil : -> NatStr
op _;_ : NatStr NatStr -> NatStr { assoc }
var L : NatStr
eq nil ;  L = L .
eq L ; nil = L .}
```

Modelul matematic: $(S = \{Nat, NatStr\}, \leq)$,

$\Sigma = \{0 :\rightarrow Nat, s : Nat \rightarrow Nat, nil :\rightarrow NatStr,$

$\quad ; : NatStr NatStr \rightarrow NatStr\}$

$\Gamma = \{\forall L._{NatStr}(nil; L = L), \forall L._{NatStr}(L; nil = L),$

$\forall L._{NatStr} \forall P._{NatStr} \forall Q._{NatStr}((L; P); Q = L; (P; Q))\}$

$((S, \leq), \Sigma, \Gamma)$ specificaţie algebrică

Exemplul 2

Date de tipul MYNATSTR sunt:

```
 nil ; s 0
 s s 0 ; s 0 ; 0
(nil ; s 0) ; s s 0 ; nil ; 0
```

Aceste date le numim expresii sau termeni.

Intuitiv: modelul matematic al unei specificaţii este o algebră de termeni; un program este un modul, adică o specificaţie; o execuţie este o rescriere în algebra de termeni asociată.

```
open MYNATSTR
set trace o n .
reduce (nil ; s 0) ; s s 0 ; nil ; 0 .
close
```

Exemplul 2

```
%MYNATSTR> reduce (nil ; s 0) ; s s 0 ; nil ; 0 .
1>[1] rule: eq (nil ; L:NatStr) = L
    { L:NatStr |-> ((s 0) ; ((s (s 0)) ; (nil ; 0))) }
1<[1] ((nil ; (s 0)) ; ((s (s 0)) ; (nil ; 0))) -->
              ((s 0) ; ((s (s 0)) ; (nil ; 0)))
1>[2] rule: eq (A1 ; (nil ; L:NatStr)) = (A1 ; L)
    { L:NatStr |-> 0, A1 |-> ((s 0) ; (s (s 0))) }
1<[2] (((s 0) ; (s (s 0))) ; (nil ; 0)) -->
      (((s 0) ; (s (s 0))) ; 0)

(((s 0) ; (s (s 0))) ; 0):NatStr
(0.000 sec for parse, 2 rewrites(0.000 sec), 13 matches)
```

Exemplul 2

```
mod! MYNATSTR{ protecting (MYNAT)
[ Nat < NatStr ]
op nil : -> NatStr
op _;_ : NatStr NatStr -> NatStr { assoc }
var L : NatStr
eq nil ;  L = L .
eq L ; nil = L . }
```

Modulul MYNATSTR defineşte $\bigcup_{k\geq 0} \mathbb{N}^k$

(secvenţele ordonate finite de numere naturale).

```
open  MYNATSTR
reduce ( 0 ; s 0) == (0 ; s 0 ; 0) .
...
(false):Bool
close
```

Exemplul 3

```
mod! MYNATSTR1{ protecting (MYNAT)
[ Nat < NatStr1 ]
op nil : -> NatStr1
op _;_ : NatStr1 NatStr1 -> NatStr1 { comm  assoc }
var L : NatStr1
eq nil ;  L = L .
eq L ; nil = L .
eq L ; L = L . }
open  MYNATSTR1
reduce ( 0 ; s 0) == (0 ; s 0 ; 0) .
...
(true):Bool
close
```

Ce defineşte MYNATSTR1 ?

Exemplul 4

```
mod! COMPLEXRAT{ protecting (RAT)
[Rat < ComplexRat ]
op _+i_ : Rat Rat -> ComplexRat
op i_ : Rat -> ComplexRat
op _+_ : ComplexRat ComplexRat -> ComplexRat {comm assoc}
var R : Rat
eq 0 +i R = i R .
eq R +i 0 = R .
... }
```

Completaţi modulul de mai sus definind adunarea şi înmulţirea numerelor complexe. Observaţi supraîncărcarea operaţiei $+$

(`+ : Rat Rat -> Rat` şi

`+ : ComplexRat ComplexRat -> ComplexRat`)

Exemplul 5

```
mod* GROUP{ [Element]
op e : -> Element
op  _+_  : Element Element -> Element { assoc }
op -_  : Element -> Element
vars x y : Element
eq - e = e .
eq e + x = x .
eq x + e = x .
eq - - x = x .
eq (- x) + x = e .
eq x + (- x)  = e .
eq - (x + y) = (- y) + (- x) . }
```

TRS canonic

O ecuaţie $l = r$ se transformă, prin orientare de la stânga la dreapta, într-o regulă de rescriere $l \to r$. Ecuaţiile modulului GROUP determină astfel un sistem de rescriere canonic (noetherian şi confluent):

- orice şir de rescrieri se termină,
- ordinea de aplicare regulilor de rescriere nu schimbă rezultatul final.

În consecinţă, o ecuaţie $t_1 = t_2$ din teoria de ordinul I a grupurilor este verificată în orice grup dacă şi numai dacă există un termen t astfel încât $t_1 \stackrel{*}{\to} t$ şi $t_2 \stackrel{*}{\to} t$.

CafeOBJ-ul poate fi utilizat ca demonstrator.

Exemplul 5

```
open GROUP
ops x1 y1 : -> Element .
reduce x1 + ( - ((- y1) + x1)) ==   y1 .

-- reduce in %GROUP :
    ((x1 + (- ((- y1) + x1))) == y1):Bool
1>[1] rule: eq (- (x:Element + y:Element))
   = ((- y) + (- x))
    { y:Element |-> x1, x:Element |-> (- y1) }
1<[1] (- ((- y1) + x1)) --> ((- x1) + (- (- y1)))
1>[2] rule: eq (- (- x:Element)) = x
    { x:Element |-> y1 }
1<[2] (- (- y1)) --> y1
```

Exemplul 5

```
1>[3] rule: eq (x:Element + ((- x) + A1))
   = (e + A1)
    { A1 |-> y1, x:Element |-> x1 }
1<[3] (x1 + ((- x1) + y1)) --> (e + y1)
1>[4] rule: eq (e + x:Element) = x
    { x:Element |-> y1 }
1<[4] (e + y1) --> y1
1>[5] rule: eq (CXU == CYU) = #!! (coerce-to-bool
(term-equational-equal cxu cyu))
    { CXU |-> y1, CYU |-> y1 }
1<[5] (y1 == y1) --> true
(true):Bool
(0.000 sec for parse, 5 rewrites(0.016 sec), 61 matches)
%GROUP> close
```

Exemplul 5

```
open GROUP
ops x1 y1 : -> Element .
reduce x1 + ( - ((- y1) + x1)) ==   y1 .

-- reduce in %GROUP :
    ((x1 + (- ((- y1) + x1))) == y1):Bool
1>[1] rule: eq (- (x:Element + y:Element))
   = ((- y) + (- x))
    { y:Element |-> x1, x:Element |-> (- y1) }
1<[1] (- ((- y1) + x1)) --> ((- x1) + (- (- y1)))
1>[2] rule: eq (- (- x:Element)) = x
    { x:Element |-> y1 }
1<[2] (- (- y1)) --> y1
```

Exemplul 5

```
1>[3] rule: eq (x:Element + ((- x) + A1))
   = (e + A1)
    { A1 |-> y1, x:Element |-> x1 }
1<[3] (x1 + ((- x1) + y1)) --> (e + y1)
1>[4] rule: eq (e + x:Element) = x
    { x:Element |-> y1 }
1<[4] (e + y1) --> y1
1>[5] rule: eq (CXU == CYU) = #!! (coerce-to-bool
(term-equational-equal cxu cyu))
    { CXU |-> y1, CYU |-> y1 }
1<[5] (y1 == y1) --> true
(true):Bool
(0.000 sec for parse, 5 rewrites(0.016 sec), 61 matches)
%GROUP> close
```

Exemplul 6

```
mod! INTER{ protecting (NAT)
[ Nat < NatStr ]
op _;_ : NatStr NatStr -> NatStr { assoc }
vars I J : Nat
ceq I ; J = J ; I if (J < I ) .
}
```


Ce face modulul INTER ?

Exemplul 6

```
mod! INTER{ protecting (NAT)
[ Nat < NatStr ]
op _;_ : NatStr NatStr -> NatStr { assoc }
vars I J : Nat
ceq I ; J = J ; I if (J < I ) .
}

open INTER .
reduce 6 ; 1 ; 0 ; 5 .
-- reduce in %INTER : ((6 ; 1) ; (0 ; 5)):NatStr
((0 ; 1) ; (5 ; 6)):NatStr
(0.000 sec for parse, 13 rewrites(0.000 sec), 28 matches)
```

Modulul INTER imlementează un algoritm de sortare.

Exemplul 6

```
CafeOBJ> select INTER .
INTER> set trace whole on
INTER> reduce 6 ; 1 ; 0 ; 5 .
-- reduce in INTER : ((6 ; 1) ; (0 ; 5)):NatStr
[1(cond)]: (0 < 1)
   --> true
[2]: ((6 ; 1) ; (0 ; 5))
---> (6 ; ((0 ; 1) ; 5))
[3(cond)]: (0 < 6)
   --> true
[4]: (6 ; ((0 ; 1) ; 5))
---> ((0 ; 6) ; (1 ; 5))
[5(cond)]: (5 < 1)
   --> false
[6(cond)]: (1 < 6)  --> true
```

Exemplul 6

```
  [7]: ((0 ; 6) ; (1 ; 5))
---> (0 ; ((1 ; 6) ; 5))
[8(cond)]: (1 < 0)
   --> false
[9(cond)]: (5 < 6)
   --> true
[10]: (0 ; ((1 ; 6) ; 5))
---> ((0 ; 1) ; (5 ; 6))
[11(cond)]: (5 < 1)
   --> false
[12(cond)]: (1 < 0)
   --> false
[13(cond)]: (6 < 5)
   --> false
((0 ; 1) ; (5 ; 6)):NatStr
```

Exemplul 7

```
mod* ELEMENT { [Element] }

mod! STACK (X :: ELEMENT) {
[EmptyStack NonEmptyStack < Stack ]
op empty : -> EmptyStack
op push : Element Stack -> NonEmptyStack
op pop_ : NonEmptyStack -> Stack
op top_ : NonEmptyStack -> Element
var E : Element
var S : Stack
eq top push (E, S) = E .
eq pop push (E, S) = S . }
```

Modulul STACK(X) crează o stivă generică.

Exemplul 7

```
open STACK
op St : -> Stack .
op El : -> Element .

reduce top(St) .
(top St):?Element

reduce top(push(El,St)) .
(El):Element

reduce top(St) == top(pop(push(El,St))) .
(true):Bool
close
```

Exemplul 8

```
mod* ELEMENT { [Element] }
mod! STACK (X :: ELEMENT) { ... }
```

Instanţiind modulul parametru X cu modulul predefinit NAT definim stivele de numere naturale.

```
view NATELEM from ELEMENT to NAT {
                              sort Element -> Nat }
module! NATSTACK {
    protecting (STACK(X <= NATELEM)) }

open NATSTACK
reduce pop(push(2,push(1 + 3,empty))) .
```

Exemplul 8

```
%NATSTACK> reduce pop(push(2,push(1 + 3,empty))) .
-- reduce in %NATSTACK :
      (pop push(2,push((1 + 3),empty))):Stack
1>[1] rule: eq [:BDEMOD] : (NN:NzNat + NM:NzNat)
   = #! (+ nn nm)
    { NN:NzNat |-> 3, NM:NzNat |-> 1 }
1<[1] (1 + 3) --> 4
1>[2] rule: eq (pop push(E:Nat,S:Stack))
   = S
    { E:Nat |-> 2, S:Stack |-> push(4,empty) }
1<[2] (pop push(2,push(4,empty))) --> push(4,empty)

(push(4,empty)):NonEmptyStack
(0.000 sec for parse, 2 rewrites(0.000 sec), 2 matches)
%NATSTACK> close
```

Concluzii

CafeOBJ este un limbaj de specificaţie bazat pe logica ecuaţională. Un program este o colecţie de module. Execuţia este o rescriere. Câteva din caracteristicile acestui limbaj sunt:

- modularizare si parametrizare,
- definirea tipurilor de date este independentă de implementare,
- extensibilitate,
- permite tratarea erorilor şi supraîncărcarea operaţiilor,
- poate fi folosit ca demonstrator.