Obligatorisk oppgave 3 – IN5100

Peter C. Ölveczky (med Paul Kobialka)

18. oktober 2023

1 Formalia

- Jobbe én og én eller to og to.
- Frist: Onsdag 8. november, 2023, 23.59 GMT +2.
- Leveres på Devilry.

2 Oppgaven: Maude-semantikk/interpret for BORNEO

Vi skal definere en eksekverbar Maude-semantikk for et meget enkelt "Java/C/IMP"-aktig imperativt "programmeringsspråk" kalt BORNEO.

BORNEO ligner på det nesten like berømte BALI-språket fra IN2100-eksamen 2023. Hovedforskjellene er at BORNEO ikke støtter tråder (selv om man gjerne kan/bør utvide BORNEO med tråder), men til gjengjeld støtter BORNEO funksjoner/prosedyrer/metoder, som gjerne selv kan kalle funksjoner/prosedyrer/metoder.

En første utvidelse av BORNEO er å legge til arrays. *Det er ikke nødvendig å definere en Maude-semantikk for arrays, men selvsagt hyggelig hvis dere gjør det.*

Et BORNEO program består av en `main {... }` "prosedyre", og en mengde funksjon/metode/prosedyre-definisjoner. Variablene deklarert i `main` er *globale* variable, og kan aksesseres og oppdateres i alle prosedyrer (med mindre en prosedyre har en lokal variabel med samme navn). Deres verdier skal være synlige når interpreten har avsluttet eksekveringen av et BORNEO-program. En prosedyre/funksjon kan også deklarere lokale variable (i tillegg til eventuelle formelle parametre, som også kan oppdateres inne i prosedyren). Dersom en slik formell parameter eller lokal variabel har samme navn som en global variabel, er det selvsagt den lokale variabelen som oppdateres/aksesseres inne i prosedyren.

Vi har for enkelhets skyld bare `int`eger variable og funksjoner (ikke engang `void`), slik at det blir lite skrivearbeid. Merk at et heltall, som 7, representeres `[7]` i BORNEO. En prosedyre/main består av (muligens) deklarasjon av variable, og setninger. En setning kan være en tilordning (*var* `:=` *intUttrykk* `;`), "if" (`if` *boolUttrykk* `then` *setninger* `else` *setninger* `fi`, eller `if` *boolUttrykk* `then` *setninger* `fi`), "while" (`while` *boolUttrykk* `do` *setninger* `od`), eller kall på funksjoner/prosedyrer (`call` f(*intUttrykk*, *intUttrykk*, ...) `;`, eller *var* `:=` `call` f(*intUttrykk*, *intUttrykk*, ...) `;`), eller `return`

intUttrykk `;`. Parameter-overføring er "call-by-value". Etter `return` skal ingen eventuelle etterfølgende setninger i prosedyren utføres.

Syntaksen til BORNEO defineres i følgende moduler, hvor et program er en term av sort `MethodDefs`:

```
fmod BORNEO-SYNTAX-WITHOUT-METHODS is protecting INT .

  *** Variables, only of type Int.
  --- Feel free to add more, maybe in test modules:
  ops a b c d e f g h i j k l m n o p q r s t u v v1 v2 v3
      x x1 x2 x3 y y1 y2 y3 z z1 z2 z3 res : -> Var [ctor] .

  --- Sets of variables:
  sorts Var VarSet .     subsort Var < VarSet .
  op noVar : -> VarSet [ctor] .
  op __ : VarSet VarSet -> VarSet [ctor assoc comm id: noVar prec 12] .

  sort VarDecl .
  op int_; : VarSet -> VarDecl [ctor prec 13 gather (e)] .

  sort MethodBody .
  op {__} : VarDecl Statements -> MethodBody [ctor prec 14 gather (e &)] .
  --- No variable declarations; pointless for 'main', but useful
  --- for other functions that may not declare local variables:
  op {_} : Statements -> MethodBody [ctor] .


  sort MainProgram .
  op main_ : MethodBody -> MainProgram [ctor] .

  --- Statements:
  sorts Statements Statement .
  subsort Statement < Statements .

  op __ : Statements Statements -> Statements [ctor assoc id: skip] .
  op skip : -> Statements [ctor] .

  op _:=_; : Var IntExp -> Statement [ctor prec 31 gather(e &)] .
  op while_do_od : BoolExp Statements -> Statement [ctor] .
  op if_then_else_fi : BoolExp Statements Statements -> Statement [ctor] .
  op if_then_fi : BoolExp Statements -> Statement [ctor] .

  --- All numbers use notation [n] to avoid overloading with built-ins:
  sorts IntExp .
  subsort Var < IntExp .
  op [_] : Int -> IntExp [ctor] .

  op _+_ : IntExp IntExp -> IntExp [ctor] .
  op _-_ : IntExp IntExp -> IntExp [ctor] .
  op _//_ : IntExp IntExp -> IntExp [ctor] .   --- integer division
  op _*_ : IntExp IntExp -> IntExp [ctor] .    --- multiplication
  op _%_ : IntExp IntExp -> IntExp [ctor] .    --- remainder

  --- Our own Booleans:
  sort Boolean .
  ops tt ff : -> Boolean [ctor] .    --- true and false
```

```
  sort BoolExp .   subsort Boolean < BoolExp .
  op _&&_ : BoolExp BoolExp -> BoolExp [ctor] .  --- and
  op _||_ : BoolExp BoolExp -> BoolExp [ctor] .  --- or
  op !_ :  BoolExp -> BoolExp [ctor] .           --- not

  op _=_ : IntExp IntExp -> BoolExp [ctor] .     --- equal
  op _!=_ : IntExp IntExp -> BoolExp [ctor] .    --- not equal
  op _>_ : IntExp IntExp -> BoolExp [ctor] .
  op _>=_ : IntExp IntExp -> BoolExp [ctor] .
  op _<_ : IntExp IntExp -> BoolExp [ctor] .
  op _<=_ : IntExp IntExp -> BoolExp [ctor] .
endfm


fmod BORNEO-SYNTAX-WITH-METHODS is
  including BORNEO-SYNTAX-WITHOUT-METHODS .

  *** We now define method declarations, as a multiset
  *** of single MethodDef definitions:

  sort MethodDef MethodDefs .
  subsort MethodDef < MethodDefs .
  op noFunc : -> MethodDefs [ctor] .
  op __ : MethodDefs MethodDefs -> MethodDefs [ctor assoc comm id: noFunc
                                               gather (e &)] .

  *** We ASSUME and REQUIRE that there is exactly one 'main' method declaration:

  subsort MainProgram < MethodDefs .

  *** Definition of a single method:

  sort MethodName .       --- Remember to populate this sort!!!

  op int_(_)_ : MethodName ParamList MethodBody -> MethodDef [ctor prec 15] .
  op int_()_ : MethodName MethodBody -> MethodDef [ctor] .  --- no paramaters

  sorts Param ParamList .
  op int_ : Var -> Param [ctor] .
  --- for simplicity no Boolean arguments; use 1 or 0 if needed

  subsort Param < ParamList .
  op nil : -> ParamList [ctor] .
  op _,_ : ParamList ParamList -> ParamList [ctor assoc id: nil] .

  *** Now we CALL procedures/functions/methods.
  --- The user (i.e., calling program/function) can either write
  ---    call func(expr1, expr2, ...);
  ---  or can write
  ---    X := call func(expr1, expr2, ...);
  ---  for some variable X.
  ---
  ---  We also allow functions that do not have parameters, in which
  ---    case the call is just   'call func();'  or  'X := call func();'
```

```
  op _:=‘call_(); : Var MethodName -> Statement [ctor prec 31] .
  op _:=‘call_(_); : Var MethodName ExpList -> Statement [ctor prec 31] .
  op call_(); :  MethodName -> Statement [ctor prec 31] .
  op call_(_); :  MethodName ExpList -> Statement [ctor prec 31] .

  sort ExpList .
  subsort IntExp < ExpList .
  op nil : -> ExpList [ctor] .
  op _,_ : ExpList ExpList -> ExpList [ctor assoc id: nil] .

  *** Return:

  op return_; : IntExp -> Statement [ctor] .
endfm
```

Then we can define some BORNEO programs as follows. Some of them are parametric:

```
fmod BORNEO-PROGRAMS is including BORNEO-SYNTAX-WITHOUT-METHODS .
  --- Some programs:
  op factorial : -> MainProgram .
  op my-gcd : Nat Nat -> MainProgram .
  op divideByTwo : Nat -> MainProgram .

  vars M N : Nat .

  eq factorial
   =
     main {
       int x z y ;
       --- x  is some input value, and y should give the result
       x := [10];         --- let's compute the factorial of 10
       z := x ;
       y := [1] ;
       while z != [0] do
          y := y * z ;
          z := z - [1] ;
       od
     } .

  eq my-gcd(M, N)
   =
     main {
        int x y r ;
        --- x and y are input values, and the gcd is in y at the end
        x := [M] ;
        y := [N] ;
        r := x % y ;
        while r >= [1] do
           x := y ;
           y := r ;
           r := x % y ;
        od
      } .

  eq divideByTwo(N)
```

```
   =
     main {
        int x y ;
        --- y should be x divided by 2
        x := [N] ;
        while x > [0] do
           if (x % [2] = [0]) then y := y + [1] ; fi
           x := x - [1] ;
        od
      } .
endfm

fmod MORE-BORNEO-PROGRAMS is including BORNEO-SYNTAX-WITH-METHODS .
  ops fib fac : -> MethodName [ctor] .
  op testFacFib : -> MethodDefs .

  eq testFacFib
   =
     main { int a b ; a := call fib([6]); b := call fac([7]); }

     int fib(int n) {
           int i j ;
           if [2] > n then return [1];
           else
             i := call fib(n - [1]);
             j := call fib(n - [2]);
             return (i + j);
           fi
        }

     int fac(int n) {
           int i ;
           if n = [0] then return [1];
           else i := call fac(n - [1]);
              return (i * n);
           fi
        } .

  op test : Nat Nat -> MethodDefs .
  vars M N : Nat .
  eq test(M, N)
   =
     main { int a b ; a := call fib([M]); b := call fac([N]); }

     int fib(int n) {
           int i j ;
           if [2] > n then return [1]; fi
           i := call fib(n - [1]);
           j := call fib(n - [2]);
           return (i + j);
        }

     int fac(int n) {
            int i ;
            if n = [0] then return [1]; fi
            i := call fac(n - [1]);
```

```
            return (i * n);
         } .
endfm

mod TEST-BORNEO-PROGRAMS is
  including BORNEO-SEMANTICS .
  including BORNEO-PROGRAMS .
  including MORE-BORNEO-PROGRAMS .
endm

rew run factorial .
rew run my-gcd(77, 21) .
rew run divideByTwo(13) .

rew run testFacFib .
rew run test(9, 11) .

mod EVEN-MORE-PROGRAMS is including TEST-BORNEO-PROGRAMS .
  op expo : Nat Nat -> MethodDefs .
  op exp : -> MethodName [ctor] .

  vars M N : Nat .
  eq expo(M, N)
   = main { int m n res ;
            m := [M] ;
            n := [N] ;
            res := call exp(m,n);
     }

     int exp(int m, int n) {
           int i j ;
           if n < [1] then return [1] ; fi
           if n = [1] then return m ; fi
           i := call exp(m, (n - [1]));
           return m * i ;
        } .

  op lecture6 : -> MethodDefs .
  op f : -> MethodName [ctor] .

  eq lecture6
   =
     main { int i n m res ;
       n := [8] ;
       m := [2] ;
       res := call f((n + m), i) ;
     }

     int f(int p, int i) {
           int n ;
           n := [23] ;
           p := p + [17] ;
           i := [9] ;
           m := p ;
           return (p + n) + (m + i);
           m := [2023];
```

```
        while n != [0] do n := [0] ; od
   } .

op recDesc : Nat Nat -> MethodDefs .
ops plussTo plussFem : -> MethodName [ctor] .

eq recDesc(M, N)
 =
   main {int k m res ;
         k := [N] ;
         m := [M] ;
         res := call plussTo(m) ;
   }

   int plussTo(int m) {
         int res ;
         if k <= [0] then return m ; fi
         k := k - [1] ;
         res := call plussFem(m + [2]) ;
         return res ;
     }

   int plussFem(int m) {
         int res ;
         if k <= [0] then return m ; fi
         k := k - [1] ;
         res := call plussTo(m + [5]) ;
         return res ;
      } .

op prime : Nat -> MethodDefs .
op delelig : -> MethodName [ctor] .
op prim : -> Var [ctor] .

eq prime(N)
 =
   main {
     int n i res prim ;
     prim := [2] ;      --- 2 er undecided, 1 er true, 0 er false
     n := [N] ;
     if n < [4] then prim := [1] ;  --- primtall, 1 er true, 0 er false
     else
       i := n - [1] ;
       while i > [1] do
         res := call delelig(n, i) ;
         if res = [1] then prim := [0] ; fi
         --- n delelig med i; ikke prim
         i := i - [1] ;
       od
     fi
     if prim = [2] then prim := [1] ; fi  ---  tallet er et primtall!
  }

   int delelig(int n, int i) {
         int res ;
         if n % i = [0] then res := [1] ; else res := [0] ; fi
```

```
            return res ;
        }  .
endm
```

Tilstandoverganger i semantikken/interpreten skal/bør modelleres av omskrivningsregler, selv om et trådløst program *bør* være terminerende og konfluent.

Du bør ha en operator `op run_ : MethodDefs -> ...` som initialiserer tilstanden til din interpret, slik at man kan eksekvere programmet *prog* ved å gi kommandoen `rew run` *prog* `.` Se filen `borneo-public.maude` på emnesiden for endel eksempler på eksekveringskommandoer dere bør kjøre.

Dere kan trygt se bort fra `gather` attributtene; gjort for parsing. Dette kan forbedres en del mer med mye mer nøyaktige operator-presendenser.

Til sist: det er nå veldig enkelt å sjekke statisk at et program er 'vel-formet' (f eks, er alle variable som brukes deklarert?). Selvsagt fint å gjøre det, men vi forlanger ikke robusthet i en slik oblig. Du kan med andre ord, som jeg, anta at alle programmene er velformede.