# BPLC Library

Christiano Braga
cbraga@ic.uff.br

Instituto de Computação
Universidade Federal Fluminense

April 20, 2018



**Abstract.** Basic Programming Languages Constructs (pronounced "beeplick") is a library of common programming language constructs whose semantics are formally specified. Therefore, to give semantics to a programming language means to relate constructions of the given language with the constructs of BPLC. It is implemented in the Maude language.

## 1 Generalized SMC machines

In [**?**], Plotkin defined Structural Operational Semantics, one of the most applied frameworks for the definition of the semantics of programming languages. Essentially, the framework consists of transition rules that specify how configurations, formed by program syntax and semantics information, such as the store or the environment, evolve in time as each construct of a given programming language is executed by a program. Thus, SOS assigns to a program a transition system where the nodes are the states of a program and the transitions denote the execution of a programming language construct that changes the state of the given program.

In this same paper, Plotkin defined yet another simple approach, called SMC machines, to the same end, but that includes more states than "necessary" and when compared to SOS. SMC machines uses a calculator metaphor in Łukasiewicz posfixed notation, aka. reverse polish notation, to execute a program. It is a stack machine with three components: a value stack, the memory store and a control stack. The semantics of a programming language is defined by rules specifying how such a machine changes as constructs are executed.

As an illustration, let us consider a programming language command, assignment for instance, that requires the evaluation of its right-hand (RHS) side before the memory location associated with its left-hand side (LHS) can be updated. First a command opcode is pushed into the control stack, such as ASSIGN, then the identifier in the LHS is pushed into the control stack, and finally the RHS expression. The evaluation of RHS should lead to a (storable) value on the top of the value stack and so the location bound to the identifier in the LHS. Once opcode ASSIGN is found in the control stack then the actual update of the appropriate memory location is performed with the value on top of the value stack.

If from the one hand there is some additional bureaucracy to transform a construct into its posfixed form, to push and pop values from the stacks, and finally yield the result, from the other hand this framework allows for the writing of rule-based specifications with a very nice characteristic: rules are unconditional. This characteristic is interesting from a proof-theoretic perspective and for the same reason it gives rise to quite efficient execution of specifications when an appropriate executable framework is chosen to animate the specification or perform analysis on it.

With this thesis, that *SMC machines are an effective means to specify and implement operational semantics of programming languages*, we *generalize it* so that a set of semantic components can be made part of the configuration, denoting the states of the associted (unlabeled) transition system.

⟨*gsmc*⟩≡

```
---------------
-- Generalized SMC machines.

fmod GSMC-SORTS is
    sorts SemComp .
endfm

view SemComp from TRIV to GSMC-SORTS is
    sort Elt to SemComp .
endv

fmod GSMC is
    ex SET{SemComp} * (op empty to noSemComp) .
    sorts Attrib Conf .
    op <_> : Set{SemComp} -> Conf [format(c! c! c! o)] .
endfm
```

## 2 Semantic components

⟨*bplc-sem-comp*⟩≡

```
fmod VALUE-SORT is
    sort Value .
endfm

fmod CONTROL-SORT is
    sort Control .
endfm

view Control from TRIV to CONTROL-SORT is
    sort Elt to Control .
endv

view Value from TRIV to VALUE-SORT is
    sort Elt to Value .
endv

-- Note: AI theories are not currently supported by Maude unification,
-- as of Alpha 115.
fmod GNELIST{X :: TRIV} is
    pr NAT .
    sorts GNeList{X} .
    subsort X$Elt < GNeList{X} .

    op __ : GNeList{X} GNeList{X} -> GNeList{X} [ctor assoc prec 25] .
endfm

-------------------------------
-- GSMC semantics for basic programming language constructs .

fmod VALUE-STACK is
    pr GNELIST{Value} * (sort GNeList{Value} to NeValueStack) .
    sort ValueStack .
    subsort NeValueStack < ValueStack .
    op evs : -> ValueStack .
    op __ : ValueStack ValueStack -> ValueStack [ditto] .
endfm

fmod CONTROL-STACK is
    pr GNELIST{Control} * (sort GNeList{Control} to NeControlStack) .
    sort ControlStack .
    subsort NeControlStack < ControlStack .
    op ecs : -> ControlStack .
    op __ : ControlStack ControlStack -> ControlStack [ditto] .
```

```
endfm

fmod STORE-SORTS is
    sorts Loc Storable .
endfm

view Loc from TRIV to STORE-SORTS is
    sort Elt to Loc .
endv

view Storable from TRIV to STORE-SORTS is
    sort Elt to Storable .
endv

fmod STORE is
    pr NAT .
    ex MAP{Loc,Storable} * (sort Entry{Loc,Storable} to Cell,
       sort Map{Loc,Storable} to Store,
       op undefined to undefloc, op empty to noStore).
    ex SET{Loc} * (op empty to noLocs) .

    op loc : Nat -> Loc [ctor] .
    op newLoc : Store -> Loc .
    op $newLoc : Store Nat -> Loc .
    eq newLoc(noStore) = loc(0) .
    ceq newLoc(S:Store) = $newLoc(S:Store, 0) if S:Store =/= noStore .
    eq $newLoc(noStore, N:Nat) = loc(N:Nat + 1) .
    ceq $newLoc((S:Store, loc(N:Nat) |-> O:Storable), N':Nat) =
        $newLoc(S:Store, N:Nat) if N:Nat >= N':Nat .
    ceq $newLoc((S:Store, loc(N:Nat) |-> O:Storable), N':Nat) =
        $newLoc(S:Store, N':Nat) if N:Nat < N':Nat .

    op $free : Loc Store -> Store .
    eq $free(L:Loc, ((L:Loc |-> O:Storable), S:Store)) = S:Store .
    eq $free(L:Loc, S:Store) = S:Store [owise] .

    op free : Set{Loc} Store -> Store .
    eq free(noLocs, S:Store) = S:Store .
    eq free((L:Loc , SL:Set{Loc}), S:Store) =
       free(SL:Set{Loc}, $free(L:Loc, S:Store)) [owise] .
endfm

fmod ENV-SORTS is
    sorts Id Bindable .
endfm
```

```
view Id from TRIV to ENV-SORTS is sort Elt to Id . endv

view Bindable from TRIV to ENV-SORTS is sort Elt to Bindable . endv

fmod ENV is
    ex MAP{Id,Bindable} * (sort Entry{Id,Bindable} to Bind,
       sort Map{Id,Bindable} to Env,
       op undefined to undefid, op empty to noEnv).
endfm
```

## 3 Basic Programming Languages Constructs

- Expressions: identifier, rational numbers, Boolean values, arithmetic and predicates. Semantic components: Identifiers require the environment and the store. The remaining constructs do not require any semantic component.
- Commands: print, assignment, sequence, choice, block, conditional, and loop. Print updates output. The remaining constructs affect only the store.
- Declarations: variables, constants, and procedures. These constructs affect the environment.
- Abnormal termination: exit. Command exit changes the semantic component that indicates abnormal termination. The behavior of command composition is also affected by this semantic component.

### 3.1 Expressions

⟨*bplc-exp*⟩≡

```
fmod EXP is
        pr QID .   pr RAT .
    ex GSMC .  ex ENV .
    ex STORE . ex CONTROL-STACK .
    ex VALUE-STACK .

    sorts Exp NzExp Pred EnvAttrib StoreAttrib ControlAttrib ValueAttrib .
    subsort EnvAttrib StoreAttrib ControlAttrib ValueAttrib < Attrib  .
    subsort Id < Exp < Control .

    -- Identifiers
    op idn : Qid -> Id [ctor format(!g o)] .

    -- Arithmetic
    op rat : Rat -> Exp [ctor format(!g o)] .
    op boo : Bool -> Exp [ctor format(!g o)] .
    op add : Exp Exp -> Exp [format(! o)] .
    op sub : Exp Exp -> Exp [format(! o)] .
    op mul : Exp Exp -> Exp [format(! o)] .
    op div : Exp Exp -> Exp [format(! o)] .

    ops ADD SUB MUL DIV : -> Control [ctor] .

    -- Boolean expressions
    op gt : Exp Exp -> Exp [format(! o)] .
    op ge : Exp Exp -> Exp [format(! o)] .
    op lt : Exp Exp -> Exp [format(! o)] .
    op le : Exp Exp -> Exp [format(! o)] .
    op eq : Exp Exp -> Exp [format(! o)] .
    op neg : Exp -> Exp [format(! o)] .
```

```
op and : Exp Exp -> Exp [format(! o)] .
op or : Exp Exp -> Exp [format(! o)] .

ops LT LE EQ NEG AND OR : -> Control [ctor] .

-- Semantic components
op env : -> EnvAttrib .
op sto : -> StoreAttrib .
op cnt : -> ControlAttrib .
op val : -> ValueAttrib .
op _:_ : EnvAttrib Env -> SemComp [ctor format(c! b! o o)] .
op _:_ : StoreAttrib Store -> SemComp [ctor format(r! b! o o)] .
op _:_ : ControlAttrib ControlStack -> SemComp [ctor format(c! b! o o)] .
op _:_ : ValueAttrib ValueStack -> SemComp [ctor format(c! b! o o)] .

op store : Rat -> Storable [ctor format(ru! o)] .
op store : Bool -> Storable [ctor format(ru! o)] .
op bind : Loc -> Bindable [ctor] .
op bind : Rat -> Bindable [ctor] .
op bind : Bool -> Bindable [ctor] .

op val : Storable -> Value [ctor] .
op val : Rat -> Value [ctor] .
op val : Bool -> Value [ctor] .
op val : Loc -> Value [ctor] .
op val : Id -> Value [ctor] .

var ... : Set{SemComp} .

eq gt(E1:Exp, E2:Exp) = neg(le(E1:Exp, E2:Exp)) .
eq ge(E1:Exp, E2:Exp) = neg(lt(E1:Exp, E2:Exp)) .

eq [rat-exp] :
    < cnt : (rat(R:Rat) C:ControlStack), val : SK:ValueStack, ... >
 =
    < cnt : C:ControlStack,
      val : (val(R:Rat) SK:ValueStack), ... > [variant] .

eq [bool-exp] :
    < cnt : (boo(B:Bool) C:ControlStack), val : SK:ValueStack, ... >
 =
    < cnt : C:ControlStack,
      val : (val(B:Bool) SK:ValueStack), ... > [variant] .

eq [add-exp] :
```

```
    < cnt : (add(E1:Exp, E2:Exp) C:ControlStack), ... >
 =
    < cnt : (E1:Exp E2:Exp ADD C:ControlStack), ... > [variant] .

eq [add-exp] :
    < cnt : (ADD C:ControlStack),
      val : (val(R1:Rat) val(R2:Rat) SK:ValueStack), ... >
 =
    < cnt : C:ControlStack,
      val : (val(R1:Rat + R2:Rat) SK:ValueStack), ... > [variant] .

eq [sub-exp] :
    < cnt : (sub(E1:Exp, E2:Exp) C:ControlStack), ... >
 =
    < cnt : (E1:Exp E2:Exp SUB C:ControlStack), ... > [variant] .

eq [sub-exp] :
    < cnt : (SUB C:ControlStack),
      val : (val(R1:Rat) val(R2:Rat) SK:ValueStack), ... >
 =
    < cnt : C:ControlStack,
      val : (val(R2:Rat - R1:Rat) SK:ValueStack), ... > [variant] .

eq [mul-exp] :
    < cnt : (mul(E1:Exp, E2:Exp) C:ControlStack), ... >
 =
    < cnt : (E1:Exp E2:Exp MUL C:ControlStack), ... > [variant] .

eq [mul-exp] :
    < cnt : (MUL C:ControlStack),
      val : (val(R1:Rat) val(R2:Rat) SK:ValueStack), ... >
 =
    < cnt : C:ControlStack,
      val : (val(R1:Rat * R2:Rat) SK:ValueStack), ... > [variant] .

eq [div-exp] :
    < cnt : (div(E1:Exp, E2:Exp) C:ControlStack), ... >
 =
    < cnt : (E1:Exp E2:Exp DIV C:ControlStack), ... > [variant] .

eq [div-exp] :
    < cnt : (DIV C:ControlStack),
      val : (val(R1:Rat) val(R2:NzRat) SK:ValueStack), ... >
 =
    < cnt : C:ControlStack,
```

```
      val : (val(R1:Rat / R2:NzRat) SK:ValueStack), ... > [variant] .

eq [lt-exp] :
    < cnt : (lt(E1:Exp, E2:Exp) C:ControlStack), ... >
 =
    < cnt : (E2:Exp E1:Exp LT C:ControlStack), ... > [variant] .

eq [lt-exp] :
    < cnt : (LT C:ControlStack),
      val : (val(R1:Rat) val(R2:Rat) SK:ValueStack), ... >
 =
    < cnt : C:ControlStack,
      val : (val(R1:Rat < R2:Rat) SK:ValueStack), ... > [variant] .

eq [le-exp] :
    < cnt : (le(E1:Exp, E2:Exp) C:ControlStack), ... >
 =
    < cnt : (E2:Exp E1:Exp LE C:ControlStack), ... > [variant] .

eq [le-exp] :
    < cnt : (LE C:ControlStack),
      val : (val(R1:Rat) val(R2:Rat) SK:ValueStack), ... >
 =
    < cnt : C:ControlStack,
      val : (val(R1:Rat <= R2:Rat) SK:ValueStack), ... > [variant] .

eq [eq-exp] :
    < cnt : (eq(E1:Exp, E2:Exp) C:ControlStack), ... >
 =
    < cnt : (E2:Exp E1:Exp EQ C:ControlStack), ... > [variant] .

eq [eq-exp] :
    < cnt : (EQ C:ControlStack),
      val : (val(R1:Rat) val(R2:Rat) SK:ValueStack), ... >
 =
    < cnt : C:ControlStack,
      val : (val(R1:Rat == R2:Rat) SK:ValueStack), ... > [variant] .

eq [eq-exp] :
    < cnt : (EQ C:ControlStack),
      val : (val(B1:Bool) val(B2:Bool) SK:ValueStack), ... >
 =
    < cnt : C:ControlStack,
      val : (val(B1:Bool == B2:Bool) SK:ValueStack), ... > [variant] .
```

```
eq [neg-exp] :
    < cnt : (neg(E:Exp) C:ControlStack), ... >
 =
    < cnt : (E:Exp NEG C:ControlStack), ... > [variant] .

eq [neg-exp] :
    < cnt : (NEG C:ControlStack),
      val : (val(B:Bool) SK:ValueStack), ... >
 =
    < cnt : C:ControlStack,
      val : (val(not(B:Bool)) SK:ValueStack), ... > [variant] .

eq [and-exp] :
    < cnt : (and(E1:Exp, E2:Exp) C:ControlStack), ... >
 =
    < cnt : (E1:Exp E2:Exp AND C:ControlStack), ... > [variant] .

eq [and-exp] :
    < cnt : (AND C:ControlStack),
      val : (val(B1:Bool) val(B2:Bool) SK:ValueStack), ... >
 =
    < cnt : C:ControlStack,
      val : (val(B1:Bool and B2:Bool) SK:ValueStack), ... > [variant] .

eq [or-exp] :
    < cnt : (or(E1:Exp, E2:Exp) C:ControlStack), ... >
 =
    < cnt : (E1:Exp E2:Exp OR C:ControlStack), ... > [variant] .

eq [and-exp] :
    < cnt : (OR C:ControlStack),
      val : (val(B1:Bool) val(B2:Bool) SK:ValueStack), ... >
 =
    < cnt : C:ControlStack,
      val : (val(B1:Bool or B2:Bool) SK:ValueStack), ... > [variant] .

eq [variable-exp] :
    < env : (I:Id |-> bind(L:Loc), E:Env),
      sto : (L:Loc |-> store(R:Rat), S:Store),
      cnt : (I:Id C:ControlStack), val : SK:ValueStack, ... >
 =
    < env : (I:Id |-> bind(L:Loc), E:Env),
      sto : (L:Loc |-> store(R:Rat), S:Store),
      cnt : C:ControlStack ,
      val : (val(R:Rat) SK:ValueStack) , ... > [variant] .
```

```
    eq [variable-exp] :
        < env : (I:Id |-> bind(L:Loc), E:Env),
          sto : (L:Loc |-> store(B:Bool), S:Store),
          cnt : (I:Id C:ControlStack), val : SK:ValueStack, ... >
     =
        < env : (I:Id |-> bind(L:Loc), E:Env),
          sto : (L:Loc |-> store(B:Bool), S:Store),
          cnt : C:ControlStack ,
          val : (val(B:Bool) SK:ValueStack) , ... > [variant] .

    eq [constant-rat-exp] :
        < env : (I:Id |-> bind(R:Rat), E:Env), cnt : (I:Id C:ControlStack),
          val : SK:ValueStack , ... >
     =
        < env : (I:Id |-> bind(R:Rat), E:Env), cnt : C:ControlStack,
          val : (val(R:Rat) SK:ValueStack), ... > [variant] .

    eq [constant-bool-exp] :
        < env : (I:Id |-> bind(B:Bool), E:Env), cnt : (I:Id C:ControlStack),
          val : SK:ValueStack, ... >
     =
        < env : (I:Id |-> bind(B:Bool), E:Env), cnt : C:ControlStack,
          val : (val(B:Bool) SK:ValueStack), ... > [variant] .
endfm
```

### 3.2 Commands

⟨*bplc-cmd*⟩≡

```
mod CMD is
    ex EXP .

    sorts Cmd ExcAttrib Exc .
    subsort Cmd < Control .

    op nop : -> Cmd [ctor format(! o)] .
    op choice : Cmd Cmd -> Cmd [ctor assoc comm format(! o)] .
    op assign : Id Exp -> Cmd [ctor format(! o)] .
    op ASSIGN : -> Control [ctor] .
    op loop : Exp Cmd -> Cmd [ctor format(! o)] .
    op LOOP : -> Control [ctor] .
    op if : Exp Cmd Cmd -> Cmd [ctor format(! o)] .
    op IF : -> Control [ctor] .
    op val : Cmd -> Value [ctor] .

    var ... : Set{SemComp} . var E : Env . var S : Store .
    var C : ControlStack . var V : ValueStack .

    eq [nop-cmd] :
        < cnt : nop C, ... > = < cnt : C, ... > [variant] .

    rl [choice-cmd] :
        < cnt : choice(M1:Cmd, M2:Cmd) C, ... > =>
        < cnt : M1:Cmd C, ... > . -- [narrowing] .

    eq [assign-cmd] :
        < env : (I:Id |-> bind(L:Loc), E),
          cnt : (assign(I:Id, E:Exp) C),
          val : V, ... >
     =
        < env : (I:Id |-> bind(L:Loc), E),
          cnt : (E:Exp ASSIGN C),
          val : (val(I:Id) V), ... > [variant] .

    eq [assign-cmd] :
        < env : (I:Id |-> bind(L:Loc), E),
          sto : (L:Loc |-> T:Storable, S),
          cnt : (ASSIGN C),
          val : (val(R:Rat) val(I:Id) V), ... >
     =
        < env : (I:Id |-> bind(L:Loc), E),
          sto : (L:Loc |-> store(R:Rat), S),
```

```
      cnt : C,
      val : V, ... > [variant] .

eq [assign-cmd] :
    < env : (I:Id |-> bind(L:Loc), E),
      sto : (L:Loc |-> T:Storable, S),
      cnt : (ASSIGN C),
      val : (val(B:Bool) val(I:Id) V), ... >
 =
    < env : (I:Id |-> bind(L:Loc), E),
      sto : (L:Loc |-> store(B:Bool), S),
      cnt : C,
      val : V, ... > [variant] .

eq [loop] :
    < cnt : loop(E:Exp, K:Cmd) C, val : V, ... >
 =
    < cnt : E:Exp LOOP C,
      val : val(loop(E:Exp, K:Cmd)) V, ... > [variant] .

rl [loop] :
    < cnt : LOOP C,
      val : val(true) val(loop(E:Exp, K:Cmd)) V, ... >
=>
    < cnt : K:Cmd loop(E:Exp, K:Cmd) C,
      val : V, ... > . -- [narrowing] .

eq [loop] :
    < cnt : LOOP C,
      val : val(false) val(loop(E:Exp, K:Cmd)) V, ... >
 =
    < cnt : C, val : V, ... > [variant] .

eq [if] :
   < cnt : if(E:Exp, K1:Cmd, K2:Cmd) C, val : V, ... >
 =
   < cnt : E:Exp IF C,
     val : val(if(E:Exp, K1:Cmd, K2:Cmd)) V, ... > [variant] .

eq [if] :
   < cnt : IF C,
     val : val(true) val(if(E:Exp, K1:Cmd, K2:Cmd)) V, ... >
 =
   < cnt : K1:Cmd C,
     val : V, ... > [variant] .
```

```
    eq [if] :
       < cnt : IF C,
         val : val(false) val(if(E:Exp, K1:Cmd, K2:Cmd)) V, ... >
     =
       < cnt : K2:Cmd C,
         val : V, ... > [variant] .
endm
```

### 3.3 Declarations

⟨*bplc-dec*⟩≡

```
mod DEC is
    ex CMD .

    sorts Abs Blk Dec Formal Formals Actual Actuals LocsAttrib .
    subsort Actuals Dec < Control .
    subsort Formal < Formals .
    subsort Exp < Actual < Actuals .
    subsort Blk < Cmd .
    subsort Abs < Bindable .

    op cns : Id Exp -> Dec [ctor format(! o)] .
    op ref : Id Exp -> Dec [ctor format(! o)] .
    op prc : Id Blk -> Dec [ctor format(! o)] .
    op prc : Id Formals Blk -> Dec [ctor format(! o)] .
    op par : Id -> Formal [ctor format(! o)] .
    op vod : -> Formal [ctor format(! o)] .
    op for : Formals Formals -> Formals [ctor assoc format(! o)] .
    op dec : Dec Dec -> Dec [ctor format(! o)] .
    op blk : Cmd -> Blk [ctor format(! o)] .
    op blk : Dec Cmd -> Blk [ctor format(! o)] .
    op cal : Id -> Cmd [ctor format(! o)] .
    op cal : Id Actuals -> Cmd [ctor format(! o)] .
    op act : Actuals Actuals -> Actuals [ctor assoc format(! o)] .
    ops CNS REF CAL BLK FRE : -> Control [ctor] .

    op val : Env -> Value [ctor] .
    op val : Loc -> Value [ctor] .
    op val : Abs -> Value [ctor] .
        op val : Set{Loc} -> Value [ctor] .

    op abs : Blk -> Abs [ctor] .
    op abs : Formals Blk -> Abs [ctor] .
    op locs : -> LocsAttrib [ctor] .
    op _:_ : LocsAttrib Set{Loc} -> SemComp [ctor format(c! b! o o)] .

    var ... : Set{SemComp} . vars E E' : Env . var S : Store .
    var C : ControlStack . var V : ValueStack .

    eq [blk] :
        < cnt : blk(D:Dec, M:Cmd) C, env : E , val : V , locs : SL:Set{Loc} , ... >
     =
        < cnt : D:Dec M:Cmd BLK C, env : E ,
                  val : val(E) val(SL:Set{Loc}) V ,
```

```
             locs : noLocs , ... > [variant] .

eq [blk] :
    < cnt : blk(M:Cmd) C, env : E , val : V , locs : SL:Set{Loc} , ... >
 =
    < cnt : M:Cmd BLK C, env : E ,
             val : val(E) val(SL:Set{Loc}) V ,
             locs : noLocs , ... > [variant] .

eq [blk] :
    < cnt : BLK C ,
      env : E' ,
      val : val(E) val(SL:Set{Loc}) V ,
     locs : SL':Set{Loc},
      sto : S:Store, ... >
 =
    < cnt : C ,
      env : E ,
      val : V ,
     locs : SL:Set{Loc},
      sto : free(SL':Set{Loc}, S:Store), ... > [variant] .

eq [ref] :
    < cnt : ref(I:Id, X:Exp) C , val : V , ... > =
    < cnt : X:Exp REF C , val : val(I:Id) V , ... > [variant] .

eq [ref] :
    < cnt : REF C, env : E ,
      sto : S ,
      val : val(R:Rat) val(I:Id) V ,
      locs : SL:Set{Loc} , ... >
 =
    < cnt : C ,
      env : insert(I:Id, bind(newLoc(S)), E) ,
      sto : insert(newLoc(S), store(R:Rat), S) ,
      val : V ,
      locs : (newLoc(S) , SL:Set{Loc}) , ... > [variant] .

eq [cns] :
    < cnt : cns(I:Id, X:Exp) C , val : V , ... > =
    < cnt : X:Exp CNS C , val : val(I:Id) V , ... > [variant] .

eq [cns] :
    < cnt : CNS C, env : E , val : val(R:Rat) val(I:Id) V , ... > =
    < cnt : C ,
```

```
    env : (I:Id |-> bind(R:Rat) , E) ,
    val : V , ... > [variant] .

eq [prc] :
  < cnt : prc(I:Id, F:Formals, B:Blk) C, env : E, ... > =
  < cnt : C,
    env : insert(I:Id, abs(F:Formals, B:Blk), E), ... > [variant] .

eq [prc] :
  < cnt : prc(I:Id, B:Blk) C, env : E, ... > =
  < cnt : C,
    env : insert(I:Id, abs(B:Blk), E), ... > [variant] .

eq [dec] :
  < cnt : dec(D1:Dec, D2:Dec) C, ... > =
  < cnt : D1:Dec D2:Dec C , ... > [variant] .

eq [cal] :
  < cnt : cal(I:Id) C, ... > =
  < cnt : I:Id CAL C, ... > [variant] .

eq [cal] :
  < cnt : cal(I:Id, A:Actuals) C, ... > =
  < cnt : I:Id A:Actuals CAL C, ... > [variant] .

eq [cal] :
  < cnt : CAL C,
    val : V1:ValueStack
    val(abs(F:Formals, B:Blk)) V2:ValueStack, ... > =
  < cnt : addDec(match(F:Formals, V1:ValueStack), B:Blk) C,
    val : V2:ValueStack, ... > [variant] .

eq [cal] :
  < cnt : CAL C,
    val : val(abs(B:Blk)) V:ValueStack, ... > =
  < cnt : B:Blk C,
    val : V:ValueStack, ... > [variant] .

eq [prc-id] :
  < cnt : (I:Id C),
    env : (I:Id |-> A:Abs, E),
    val : V , ... > =
  < cnt : C,
    env : (I:Id |-> A:Abs, E),
    val : (val(A:Abs) V), ... > [variant] .
```

```
    eq [act] :
        < cnt : act(E:Exp, A:Actuals) C, ... > =
        < cnt : A:Actuals E:Exp C, ... > [variant] .

    op match : Formals ValueStack -> Dec .
    eq match(par(I:Id), val(R:Rat)) = ref(I:Id, rat(R:Rat)) .
    eq match(par(I:Id), val(B:Bool)) = ref(I:Id, boo(B:Bool)) .
    eq match(for(F:Formal, L:Formals), (V:Value VS:ValueStack)) =
        dec(match(F:Formal, V:Value),
          match(L:Formals, VS:ValueStack)) .

    op addDec : Dec Blk -> Blk .
    eq addDec(D:Dec, B:Blk) = blk(D:Dec, B:Blk) .
endm
```

### 3.4 Output

⟨*bplc-out*⟩≡

```
mod OUT is
    ex DEC .

    sort OutAttrib .

    op out : -> OutAttrib [ctor] .
    op _:_ : OutAttrib ValueStack -> SemComp [ctor format(c! b! o o)] .
    op print : Exp -> Cmd [ctor format(! o)] .
    op PRINT : -> Control [ctor] .

    var ... : Set{SemComp} .

    op out : Conf -> NeValueStack [ctor] .
    eq out(< out : V:NeValueStack , ... >) = V:NeValueStack .

    eq [print] :
        < cnt : (print(E:Exp) C:ControlStack), ... > =
        < cnt : (E:Exp PRINT C:ControlStack), ... > [variant] .

    eq [print] :
        < cnt : (PRINT C:ControlStack),
          val : val(R:Rat) V:ValueStack,
          out : O:ValueStack, ... > =
        < cnt : C:ControlStack,
          val : V:ValueStack,
          out : val(R:Rat) O:ValueStack , ... > [variant] .
endm
```

### 3.5 Abnormal termination

⟨*bplc-ext*⟩≡

```
mod EXIT is
    ex OUT .

    -- Sequences had to be moved "down" from CMD to this module because
    -- the evaluation of the next command only takes place if no
    -- exit was executed.
    op seq : Cmd Cmd -> Cmd [format(! o)] .

    op exit : Exp -> Cmd [format(! o)] .
    op EXT : -> Exc .
    op EXIT : -> Control .
    op CNT : -> Exc .
    op exc : -> ExcAttrib .
    op _:_ : ExcAttrib Exc -> SemComp [format(c! b! o o)] .

    var ... : Set{SemComp} . var E : Env . var S : Store .
    var C : ControlStack . var V : ValueStack .

    eq [exit-cmd] :
        < cnt : exit(X:Exp) C, ... > = < cnt : X:Exp EXIT C, ... > [variant] .

    -- Maybe define a flush operation that sets the semantic components
    -- to their identity values.
    eq [exit-cmd] :
        < cnt : EXIT C,
          env : E,
          sto : S,
          val : A:Value V,
          out : O:ValueStack,
          locs : SL:Set{Loc},
          exc : CNT, ... > =
        < cnt : ecs,
          env : noEnv,
          sto : noStore,
          val : evs,
          out : A:Value,
          locs : noLocs,
          exc : EXT, ... > [variant] .

    -- Sequences had to be moved "down" to this module because
    -- the evaluation of the next command only takes place if no
    -- exit was executed.
    eq [seq-cmd] :
```

```
        < cnt : seq(C1:Cmd, C2:Cmd) C, exc : CNT, ... > =
        < cnt : C1:Cmd C2:Cmd C, exc : CNT, ... > [variant] .

    eq [seq-cmd] :
        < cnt : seq(C1:Cmd, C2:Cmd) C, exc : EXT, ... > =
        < cnt : ecs, exc : EXT, ... > [variant] .
endm
```

### 3.6 Running a command

⟨*bplc*⟩≡

```
mod BPLC is
    ex EXIT .
    var ... : Set{SemComp} .

    op getValue : Id Conf -> Storable .
    eq getValue(I:Id,
          < env : (I:Id |-> bind(L:Loc), E:Env) ,
          sto : (L:Loc |-> S:Storable, S:Store) ,
          ... >) = S:Storable .

        op run : Cmd Dec -> Conf .
        rl run(C:Cmd, D:Dec) =>
                < cnt : blk(D:Dec, C:Cmd) ecs,
                  env : noEnv, sto : noStore , val : evs ,
                  locs : noLocs , out : evs , exc : CNT > .
endm
```

## 4 Model checking

⟨*bplc-mc*⟩≡

```
mod BPLC+META-LEVEL is
        pr BPLC . pr META-LEVEL .
endm

load model-checker

mod BPLC-MODEL-CHECKER is
    ex BPLC .
    pr MODEL-CHECKER .
        pr LTL-SIMPLIFIER .
        pr META-LEVEL .

    subsort Conf < State .

    var ... : Set{SemComp} .

    op valueOf : Id Rat Conf -> Bool .
    op valueOf : Id Bool Conf -> Bool .
    eq valueOf(I:Id, R:Rat,
          < env : (I:Id |-> bind(L:Loc), E:Env) ,
            sto : (L:Loc |-> store(R:Rat), S:Store) ,
            ... >) = true .
    eq valueOf(I:Id, R:Rat, C:Conf) = false [owise] .

    eq valueOf(I:Id, B:Bool,
          < env : (I:Id |-> bind(L:Loc), E:Env) ,
            sto : (L:Loc |-> store(B:Bool), S:Store) ,
            ... >) = true .
    eq valueOf(I:Id, B:Bool, C:Conf) = false [owise] .

    op _,_|=?_ : Dec Cmd Formula -> ModelCheckResult .
    eq D:Dec, C:Cmd |=? F:Formula =
           modelCheck(run(C:Cmd,D:Dec), F:Formula) .

    op makeOpDeclSet : Dec -> OpDeclSet .
    eq makeOpDeclSet(ref(idn(Q:Qid), E:Exp)) =
           (op Q:Qid : 'Rat -> 'Prop [none] .) .
    eq makeOpDeclSet(dec(D1:Dec, D2:Dec)) =
           makeOpDeclSet(D1:Dec) makeOpDeclSet(D2:Dec) .
    eq makeOpDeclSet(D:Dec) = none [owise] .

    op makeEqSet : Dec -> EquationSet .
    eq makeEqSet(ref(idn(Q:Qid), E:Exp)) =
```

```
          (ceq '_|=_['S:State, Q:Qid['R:Rat]] = 'true.Bool
           if 'valueOf['idn[upTerm(Q:Qid)],'R:Rat,'S:State] = 'true.Bool [none] .) .
  eq makeEqSet(dec(D1:Dec, D2:Dec)) =
          makeEqSet(D1:Dec) makeEqSet(D2:Dec) .
  eq makeEqSet(D:Dec) = none [owise] .

  op makeMCModule : Qid Dec -> SModule .
  eq makeMCModule(Q:Qid, D:Dec) =
     (mod qid("MODEL-CHECK-" + string(Q:Qid)) is
                    protecting 'BPLC-MODEL-CHECKER .
           sorts none .
           none -- SubsortDeclSet
           -- op 'bid : 'Qid -> 'GSLIdentifiers [none] .
           makeOpDeclSet(D:Dec)
           none -- MembAxSet
           makeEqSet(D:Dec)
           none endm) .
endm
```

## 5 Generating Python code

⟨*bplc-compile-to-python*⟩≡

```
mod COMPILE-TO-PYTHON is
    pr BPLC .
    pr QID-LIST .
    pr META-LEVEL .

    ops level incr : -> Nat .
    eq level = 3 .
    eq incr = 3 .

    op printSpaces : Nat -> QidList .
    eq printSpaces(0) = (nil).QidList .
    eq printSpaces(N:Nat) = '\s printSpaces(sd(N:Nat, 1)) .

    op id2Qid : Id -> QidList .
    eq id2Qid(idn(Q:Qid)) = Q:Qid .

    op formals2QidList : Formals -> QidList .
    eq formals2QidList(vod) = (nil).QidList .
    eq formals2QidList(par(I:Id)) = id2Qid(I:Id) .
    eq formals2QidList(for(F1:Formals, F2:Formals)) =
       formals2QidList(F1:Formals) '`, formals2QidList(F2:Formals) .

    op blk2Block : Blk Dec Nat -> QidList .
   ceq blk2Block(blk(D1:Dec, C:Cmd), D2:Dec, N:Nat) =
       '\s ':
       declareGlobal(C:Cmd, D2:Dec, N:Nat + incr)
       dec2Decl(D1:Dec, D2:Dec, N:Nat + incr)
       cmd2Comm(C:Cmd, D2:Dec, N:Nat + incr)
    if declareGlobal(C:Cmd, D2:Dec, N:Nat + incr) =/= (nil).QidList .
   ceq blk2Block(blk(D1:Dec, C:Cmd), D2:Dec, N:Nat) =
        '\s ': '\n
        dec2Decl(D1:Dec, D2:Dec, N:Nat + incr)
        cmd2Comm(C:Cmd, D2:Dec, N:Nat + incr)
    if declareGlobal(C:Cmd, D2:Dec, N:Nat + incr) == (nil).QidList .
   ceq blk2Block(blk(C:Cmd), D:Dec, N:Nat) =
       '\s ':
       declareGlobal(C:Cmd, D:Dec, N:Nat + incr)
       cmd2Comm(C:Cmd, D:Dec, N:Nat + incr)
    if declareGlobal(C:Cmd, D:Dec, N:Nat + incr) =/= (nil).QidList .
   ceq blk2Block(blk(C:Cmd), D:Dec, N:Nat) =
        '\s ':
        cmd2Comm(C:Cmd, D:Dec, N:Nat + incr)
    if declareGlobal(C:Cmd, D:Dec, N:Nat + incr) =/= (nil).QidList .
```

```
op dec2Decl : Dec Dec Nat -> QidList .
eq dec2Decl(cns(I:Id, E:Exp), D:Dec, N:Nat) =
   id2Qid(I:Id) '= exp2Expr(E:Exp) .
eq dec2Decl(ref(I:Id, E:Exp), D:Dec, N:Nat) =
   id2Qid(I:Id) '= exp2Expr(E:Exp) .
eq dec2Decl(prc(I:Id, B:Blk), D:Dec, N:Nat) =
   'def id2Qid(I:Id) '`( '`)
   blk2Block(B:Blk, D:Dec, N:Nat + incr) .
eq dec2Decl(prc(I:Id, F:Formals, B:Blk), D:Dec, N:Nat) =
   'def id2Qid(I:Id) '`( formals2QidList(F:Formals) '`)
   blk2Block(B:Blk, D:Dec, N:Nat + incr) .
eq dec2Decl(dec(D1:Dec, D2:Dec), D3:Dec, N:Nat) =
   dec2Decl(D1:Dec, D3:Dec, N:Nat) '\n
   dec2Decl(D2:Dec, D3:Dec, N:Nat) .

op isGlobal : Id Dec -> Bool [memo] .
eq isGlobal(I1:Id, ref(I2:Id, E:Exp)) = I1:Id == I2:Id .
eq isGlobal(I1:Id, cns(I2:Id, E:Exp)) = false .
eq isGlobal(I1:Id, prc(I2:Id, F:Formals, B:Blk)) = false .
eq isGlobal(I1:Id, prc(I2:Id, B:Blk)) = false .
eq isGlobal(I:Id, dec(D1:Dec, D2:Dec)) =
   isGlobal(I:Id, D1:Dec) or isGlobal(I:Id, D2:Dec) .

op exp2Expr : Exp -> QidList .
eq exp2Expr(I:Id) = id2Qid(I:Id) .
eq exp2Expr(rat(R:Rat)) =
   metaPrettyPrint(upModule('RAT, false), upTerm(R:Rat)) .
eq exp2Expr(boo(B:Bool)) =
   metaPrettyPrint(upModule('BOOL, false), upTerm(B:Bool)) .
eq exp2Expr(add(E1:Exp, E2:Exp)) =
   exp2Expr(E1:Exp) '+ exp2Expr(E2:Exp) .
eq exp2Expr(mul(E1:Exp, E2:Exp)) =
   exp2Expr(E1:Exp) '* exp2Expr(E2:Exp) .
eq exp2Expr(sub(E1:Exp, E2:Exp)) =
   exp2Expr(E1:Exp) '- exp2Expr(E2:Exp) .
eq exp2Expr(div(E1:Exp, E2:Exp)) =
   exp2Expr(E1:Exp) '/ exp2Expr(E2:Exp) .
eq exp2Expr(neg(E:Exp)) =
   'not '`( exp2Expr(E:Exp) '`) .
eq exp2Expr(eq(E1:Exp, E2:Exp)) =
   exp2Expr(E1:Exp) '== exp2Expr(E2:Exp) .
eq exp2Expr(and(E1:Exp, E2:Exp)) =
   exp2Expr(E1:Exp) 'and exp2Expr(E2:Exp) .
eq exp2Expr(or(E1:Exp, E2:Exp)) =
```

```
       exp2Expr(E1:Exp) 'or exp2Expr(E2:Exp) .
    eq exp2Expr(gt(E1:Exp, E2:Exp)) =
       exp2Expr(E1:Exp) '> exp2Expr(E2:Exp) .
    eq exp2Expr(ge(E1:Exp, E2:Exp)) =
       exp2Expr(E1:Exp) '>= exp2Expr(E2:Exp) .
    eq exp2Expr(lt(E1:Exp, E2:Exp)) =
       exp2Expr(E1:Exp) '< exp2Expr(E2:Exp) .
    eq exp2Expr(le(E1:Exp, E2:Exp)) =
       exp2Expr(E1:Exp) '<= exp2Expr(E2:Exp) .

    op cmd2Comm : Cmd Dec Nat -> QidList .
    eq cmd2Comm(print(E:Exp), D:Dec, N:Nat) =
       'print '`( exp2Expr(E:Exp) '`) .
    eq cmd2Comm(assign(I:Id, E:Exp), D:Dec, N:Nat) =
       id2Qid(I:Id) '= exp2Expr(E:Exp) .
    eq cmd2Comm(if(E:Exp, B1:Blk, B2:Blk), D:Dec, N:Nat) =
       'if exp2Expr(E:Exp)
       blk2Block(B1:Blk, D:Dec, N:Nat + incr)
       'else
       blk2Block(B2:Blk, D:Dec, N:Nat + incr) .
    eq cmd2Comm(loop(E:Exp, B:Blk), D:Dec, N:Nat) =
       'while exp2Expr(E:Exp)
       blk2Block(B:Blk, D:Dec, N:Nat) .
    eq cmd2Comm(seq(C1:Cmd, C2:Cmd), D:Dec, N:Nat) =
       printSpaces(N:Nat)
       cmd2Comm(C1:Cmd, D:Dec, N:Nat) '\n
       printSpaces(N:Nat)
       cmd2Comm(C2:Cmd, D:Dec, N:Nat) .

    op declareGlobal : Cmd Dec Nat -> QidList .
   ceq declareGlobal(assign(I:Id, E:Exp), D:Dec, N:Nat) =
       '\n printSpaces(N:Nat) 'global id2Qid(I:Id)
    if isGlobal(I:Id, D:Dec) .
    eq declareGlobal(seq(C1:Cmd, C2:Cmd), D:Dec, N:Nat) =
       declareGlobal(C1:Cmd, D:Dec, N:Nat) '\n
       declareGlobal(C2:Cmd, D:Dec, N:Nat) .
    eq declareGlobal(C:Cmd, D:Dec, N:Nat) = (nil).Qid [owise] .
endm
```