Denotational Semantics of MicroScala

Abstract Syntax

Syntax Domains:	**Abstract Syntax Rules:**
C: Compilation Unit	**C ::= D M** | **M**
M: Main Definition	**M ::= V S** | **S**
D: Definition	**D ::= D_1 ; D_2** | **Fid (P) : T = F** | **V**
V: Var Definition	**V ::= V_1 ; V_2** | **Vid : T = L**
T: Type	**T ::= Int** | **List**
P: Formal Parameters	**P ::= P_1 Id : T** | ε
F: Function Body	**F ::= V S ; return E** | **S ; return E** | **V return E** | **return E**
S: Statement	**S ::= S_1 ; S_2** | **Vid = E** | **if E S** | **if E S_1 else S_2** |**while E S** | **println E**
E: Expression	**E ::= E_1 || E_2** | **E_1 && E_2** | **E_1 < E_2** | **E_1 <= E_2** | **E_1 > E_2** | **E_1 >= E_2** | **E_1 == E_2** | **E_1 != E_2** | **+E** | **-E** | **E_1 + E_2** | **E_1 - E_2** | **E_1 * E_2** | **E_1 / E_2** | **E_1 :: E_2** | **E . head** | **E . tail** | **E . isEmpty** | **Vid** | **Fid A** | **L**
A: Arguments	**A ::= A E** | ε
L: Literal	**L ::= N** | **Nil**
Vid: Variable Id	
Fid: Function Id	
N: Integer	

Semantic Algebra:

Domain n ∈ N = {. . . , −2, −1, 0, 1, 2, . . .} (Integers)

AO [[+]] $n_1\ n_2 = (n_1 + n_2)$
AO [[−]] $n_1\ n_2 = (n_1 - n_2)$
AO [[*]] $n_1\ n_2 = (n_1 \times n_2)$
AO [[/]] $n_1\ n_2 =$ if $(n_2 = 0)$ then $\top$ else (n_1 / n_2)

RO [[<]] $n_1\ n_2 =$ if $(n_1 < n_2)$ then *true* else *false*
RO [[<=]] $n_1\ n_2 =$ if $(n_1 \leq n_2)$ then *true* else *false*
RO [[>]] $n_1\ n_2 =$ if $(n_1 > n_2)$ then *true* else *false*
RO [[>=]] $n_1\ n_2 =$ if $(n_1 \geq n_2)$ then *true* else *false*
RO [[==]] $n_1\ n_2 =$ if $(n_1 = n_2)$ then *true* else *false*
RO [[!=]] $n_1\ n_2 =$ if $(n_1 \neq n_2)$ then *true* else *false*

Domain p ∈ List = N * (List of integers)

nil	*nil* returns an empty list i.e., <| |>
cons : $n \rightarrow p_1 \rightarrow p_2$	*cons* takes an integer n and list p_1 ; attaches n to the head of p_1 and returns the resultant list p_2
hd : $p \rightarrow n$	*hd* takes a list p and returns the head of p
tl : $p_1 \rightarrow p_2$	*tl* takes the list p_1 and returns the tail of p_1 as list p_2
append : $n \rightarrow p_1 \rightarrow p_2$	*append* takes an integer n and list p_1 ; attaches n to the end of p_1 and returns the resultant list p_2
eqlist : List × List → Integer	*eqlist* compares the two lists for equality
neqlist : List × List → Integer	*neqlist* compares the two lists for inequality

Domain varid ∈ Variable Id

Domain funcid ∈ Function Id

Domain id ∈ Id = Variable Id + Function Id

Domain t ∈ Type = { *integer, boolean, list* }

Domain v ∈ Value = N + List

Domain e ∈ Expressible-value = (Type × Value°)

Expr-value : Type → Value° → Expressible-value
Exprval t v = (t, v) "constructs the expressible value tuple"

type : Expressible-value → Type	*value* : Expressible-value → Value°
type e = (e ↓ 1)	*value* e = (e ↓ 2)
"selects the type component"	"selects the value component"

Domain fb ∈ Function-body = Conf → (Conf × Expressible-value)

Domain f ∈ Function-denot = Formal Parameters × Type × Function-body

Function-denot : Formal Parameters → Type → Function-body → Function-denot

Function-denot fp t fb = (fp, t, fb) "constructs the function denotation triple"

formpars : Function-denot → Formal Parameters

formpars f = (f ↓ 1) "selects the formal parameter declarations"

type : Function-denot → Type | *funcbody* : Function-denot → Function-body

type f = (f ↓ 2) | *funcbody* f = (f ↓ 3)

"selects the type component" | "selects the function body component"

Domain d ∈ Denotable-value = Expressible-value + Function-denot

Domain c ∈ Conf = {(Env × Env × File)}˚

Conf : Env → Env → File → Conf

Conf env_{global} env_{local} fl_{out} = (env_{global}, env_{local}, fl_{out}) "constructs the configuration tuple"

global-env : Conf → Env | *local-env* : Conf → Env | *outfile* : Conf → File

env c = (c ↓ 1) | *env* c = (c ↓ 2) | *outfile* c = (c ↓ 3)

"selects the environment components" | "selects the output file component"

Domain fl ∈ File = List

Domain env ∈ Env = {(Id→Denotable-value)}˚

new-env : Env

new-env = λid . ⊥

access-env : Id → Env → Denotable-value

access-env id env = env id

update-env : Id → Denotable-value → Env → Env

update-env id d env =

  let d = *access-env* id env

  in if d ≠ ⊥ then ⊤ else env [d / id] "error if already defined"

update-val : Variable Id → Expressible-value → Env → Env

update-val varId e env =

  let d = *access-env* varId env

  in if d = ⊥ then ⊤ else env [e / id] "error if not already defined"

Semantic Functions:

C : Compilation Unit → $\text{File}_{\text{output}}$
M : Main Definition → Env → Conf
D : Definition → Env → Env
V : Var Definition → Env → Env
F : Function Body → Conf → (Conf × Expressible-value)
S : Statement → Conf → Conf
E : Expression → Conf → (Conf × Expressible-value)
P: Formal Parameters → Env → Arguments → Conf → (Env × Conf)
L : Literal → Expressible-value

Semantic Equations:

COMPILATION UNIT:

C **[[D M]]** =
    let $\text{env}_{\text{global}}$ = D **[[D]]** *new-env*
    in *outfile* (M **[[M]]** $\text{env}_{\text{global}}$)

C **[[M]]** =
    let $\text{env}_{\text{global}}$ = *new-env*
    in *outfile* (M **[[M]]** $\text{env}_{\text{global}}$)

MAIN PROGRAM:

M **[[V S]]** $\text{env}_{\text{global}}$ =
    let $\text{env}_{\text{local}}$ = V **[[V]]** *new-env*
    in S **[[S]]** (*Conf* $\text{env}_{\text{global}}$ $\text{env}_{\text{local}}$ *nil*)

M **[[S]]** $\text{env}_{\text{global}}$ =
    let $\text{env}_{\text{local}}$ = *new-env*
    in S **[[S]]** (*Conf* $\text{env}_{\text{global}}$ $\text{env}_{\text{local}}$ *nil*)

DEFINITION:

D **[[D_1 ; D_2]]** env = D **[[D_2]]** (D **[[D_1]]** env)

D **[[Fid (P) : T = F]]** env = let d = *funcDenot* **[[P]]** (T **[[T]]**) (F **[[F]]**)
    in (*update-env* **[[Fid]]** d env)

D **[[V]]** env = V **[[V]]** env

VAR DEFINITION:

V **[[V_1 ; V_2]]** env = V **[[V_2]]** (V **[[V_1]]** env)

V **[[Vid : T = L]]** env =
    let t = (T **[[T]]**)
    in let e = (L **[[L]]**)
        in if (t = *integer* and *type* e = *integer* and *value* e = 0) or
                (t = *list* and *type* e = *list* and *value* e = *nil*) then
            update-env **[[Vid]]** e env
          else ⊤ "type error"

TYPE:

T **[[Int]]** = *integer*
T **[[List]]** = *list*

FUNCTION BODY:

F **[[V S ; return E]]** c =
    let env_{local} = V **[[V]]** *new-env*
    in let c_1 = S **[[S]]** (*Conf* (*global-env* c) env_{local} (*outfile* c))
        in E **[[E]]** c_1

F **[[S ; return E]]** c =
    let env_{local} = *new-env*
    in let c_1 = S **[[S]]** (*Conf* (*global-env* c) env_{local} (*outfile* c))
        in E **[[E]]** c_1

F **[[V return E]]** c =
    let env_{local} = V **[[V]]** *new-env*
    in let c_1 = *Conf* (*global-env* c) env_{local} (*outfile* c)
        in E **[[E]]** c_1

F **[[return E]]** c =
    let env_{local} = *new-env*
    in let c_1 = *Conf* (*global-env* c) env_{local} (*outfile* c)
        in E **[[E]]** c_1

STATEMENT:

S **[[S_1 ; S_2]]** c = S **[[S_2]]** (S **[[S_1]]** c)

S **[[Vid = E]]** c = let d_{local} = *access-env* **[[Vid]]** (*local-env* c)
in if $d_{local} \neq \perp$ then
let (c_1, e) = E **[[E]]** c
in if (*type* d ≠ *type* e) then ┬ "type error"
else let env_{local} = (*update-val* **[[Vid]]** e (*local-env* c_1))
in (*Conf* (*global-env* c) env_{local} (*outfile* c_1))
else
let d_{global} = *access-env* **[[Vid]]** (*global-env* c)
in if $d_{global} \neq \perp$ then
let (c_1, e) = E **[[E]]** c
in if (*type* d ≠ *type* e) then ┬ "type error"
else let env_{global} =
(*update-val* **[[Vid]]** e (*global-env* c_1))
in (*Conf* env_{global} (*local-env* c_1) (*outfile* c_1))
else ┬ "undeclared variable error"

S **[[if E S]]** c = let (c_1, e) = E **[[E]]** c
in if (*type* e ≠ *boolean*) then ┬ "type error"
else if (*value* e) then S **[[S]]** c_1
else c_1

S **[[if E S_1 else S_2]]** c = let (c_1, e) = E **[[E]]** c
in if (*type* e ≠ *boolean*) then ┬ "type error"
else if (*value* e) then S **[[S_1]]** c_1
else S **[[S_2]]** c_1

S **[[while E S]]** c = let (c_1, e) = E **[[E]]** c
in if (*type* e ≠ *boolean*) then ┬ "type error"
else if (*value* e) then S **[[while E S]]** (S **[[S]]** c_1)
else c_1

S **[[println E]]** c = let (c_1, e) = (E **[[E]]** c)
in if (*type* e ≠ *integer*) then ┬ "type error"
else let fl = (*outfile* c_1)
in let fl_1 = (*append* (*value* e) fl)
in *Conf* (*global-env* c_1) (*local-env* c_1) fl_1

EXPRESSIONS:

E **[[E$_1$ && E$_2$]]** c =
    let (c$_1$, e$_1$) = *E* **[[E$_1$]]** c
    in if (*type* e$_1$ ≠ *boolean*) then ⊤ "type error"
        else if (*value* e$_1$) then
            let (c$_2$, e$_2$) = *E* **[[E$_2$]]** c$_1$
            in if (*type* e$_2$ ≠ *boolean*) then ⊤ "type error"
                else (c$_2$, *value* e$_2$)
        else (c$_1$, *value* e$_1$)

E **[[E$_1$ || E$_2$]]** c =
    let (c$_1$, e$_1$) = *E* **[[E$_1$]]** c
    in if (*type* e$_1$ ≠ *boolean*) then ⊤ "type error"
        else if (*value* e$_1$) then (c$_1$, *value* e$_1$)
        else let (c$_2$, e$_2$) = *E* **[[E$_2$]]** c$_1$
            in if (*type* e$_2$ ≠ *boolean*) then ⊤ "type error"
                else (c$_2$, *value* e$_2$)

E **[[!E]]** c =
    let (c$_1$, e$_1$) = *E* **[[E]]** c
    in if (*type* e$_1$ ≠ *boolean*) then ⊤ "type error"
        else (c$_1$, *not* (*value* e$_1$))

E **[[E$_1$ == E$_2$]]** c =
    let (c$_1$, e$_1$) = *E* **[[E$_1$]]** c
    in let (c$_2$, e$_2$) = *E* **[[E$_2$]]** c$_1$
        in if (*type* e$_1$ = *integer*) and (*type* e$_2$ = *integer*) then
              Exprval boolean (*RO* **[[==]]** (*value* e$_1$) (*value* e$_2$))
            else if (*type* e$_1$ = *list*) and (*type* e$_2$ = *list*) then
              Exprval boolean (*eqlist* (*value* e$_1$) (*value* e$_2$))
            else ⊤ "type error"

E **[[E$_1$!= E2]]** c =
    let (c$_1$, e$_1$) = *E* **[[E$_1$]]** c
    in let (c$_2$, e$_2$) = *E* **[[E$_2$]]** c$_1$
        in if (*type* e$_1$ = *integer*) and (*type* e$_2$ = *integer*) then
              Exprval boolean (*RO* **[[!=]]** (*value* e$_1$) (*value* e$_2$))
            else if (*type* e$_1$ = *list*) and (*type* e$_2$ = *list*) then
              Exprval boolean (*neqlist* (*value* e$_1$) (*value* e$_2$))
            else ⊤ "type error"

E [[$\mathbf{E_1}$ **rop** $\mathbf{E_2}$]] c =
    let (c_1, e_1) = E [[$\mathbf{E_1}$]] c
    in let (c_2, e_2) = E [[$\mathbf{E_2}$]] c_1
        in if (*type* $e_1 \neq$ *integer*) or (*type* $e_2 \neq$ *integer*) then $\top$ "type error"
            else let e_{result} = *Exprval boolean* (*RO*[[**rop**]] (*value* e_1) (*value* e_2))
                  in (c_2, e_{result})
    where **rop** ∈ {< , > , <= , >=}

E [[**aop** **E**]] c =
    let (c_1, e_1) = E [[**E**]] c
    in if (*type* $e_1 \neq$ *integer*) then $\top$ "type error"
        else let e_{result} = *Exprval integer* (*AO*[[**aop**]] 0 (*value* e_1))
              in (c_1, e_{result})
    where **aop** ∈ { + , - }

E [[$\mathbf{E_1}$ **aop** $\mathbf{E_2}$]] c =
    let (c_1, e_1) = E [[$\mathbf{E_1}$]] c
    in let (c_2, e_2) = E [[$\mathbf{E_2}$]] c_1
        in if (*type* $e_1 \neq$ *integer*) or (*type* $e_2 \neq$ *integer*) then $\top$ "type error"
            else let e_{result} = *Exprval integer* (*AO*[[**aop**]] (*value* e_1) (*value* e_2))
                  in (c_2, e_{result})
    where **aop** ∈ { + , - , * , / }

E [[$\mathbf{E_1}$ :: $\mathbf{E_2}$]] c =
    let (c_1, e_1) = (E [[$\mathbf{E_1}$]] c)
    in let (c_2, e_2) = (E [[$\mathbf{E_2}$]] c_1)
        in if (*type* $e_1 \neq$ *integer*) or (*type* $e_2 \neq$ *list*) then $\top$ "type error"
            else (c_2, *Exprval list* (*cons* (*value* e_1) (*value* e_2)))

E [[**E . head**]] c = let (c_1, e_1) = (E [[**E**]] c)
                          in if (*type* $e_1 \neq$ *list*) or (*value* e_1 = *nil*) then $\top$ "error"
                              else (c_1, *Exprval integer* (*hd* (*value* e_1)))

E [[**E . tail**]] c = let (c_1, e_1) = (E [[**E**]] c)
                          in if (*type* $e_1 \neq$ *list*) or (*value* e_1 = *nil*) then $\top$ "error"
                              else (c_1, *Exprval list* (*tl* (*value* e_1)))

E [[**E . isEmpty**]] env c =
    let (c_1, e_1) = (E [[**E**]] env c)
    in if (*type* $e_1 \neq$ *list*) then $\top$ "type error"
        else (c_1, *Exprval boolean* (*value* e_1 = *nil*))

E **[[Vid]]** c =
    let d_{local} = *access-env* **[[Vid]]** (*local-env* c)
    in if $d_{local} \neq \perp$ then (c, d_{local})
        else let d_{global} = *access-env* **[[Vid]]** (*global-env* c)
              in if $d_{global} \neq \perp$ then (c, d_{global})
                  else $\top$ "undeclared variable error"

E **[[L]]** c = (c, *L* **[[L]]**)

E **[[Fid A]]** c =
    let d = *access-env* **[[Fid]]** (*global-env* c)
    in if d = $\perp$ then "undeclared function error"
       else let (env_{local}, c_1) = *PM* (*formpar* d) *new-env* **[[A]]** c "**evaluate actuals"**
            in let (c_2, e_{result}) =
                  F (*funcbody* d) (*Conf* (*global-env* c_1) env_{local} (*outfile* c_1)) "**call"**
               in (*Conf* (*global-env* c_2) (*local-env* c_1) (*outfile* c_2), e_{result}) "**restore caller env"**

PARAMETER MATCHING:

P **[[T Vid P]]** env_{local} **[[E A]]** c =
    let (c_1, e) = *E* **[[E]]** c
    in if (*type* e ≠ (*T* **[[T]]**)) then $\top$ "type error"
        else let env_1 = *V* **[[T Vid]]** env_{local} "**add formals into local env"**
              in let env_2 = (*update-val* **[[Vid]]** e env_1) **"set formals to actuals"**
                  in *P* **[[P]]** env_2 **[[A]]** c_1

P **[[ ]]** env_{local} **[[ ]]** c = (env_{local}, c)

P **[[ ]]** env_{local} **[[E A]]** c = $\top$ "number of parameters mismatch"

P **[[T Vid P]]** env_{local} **[[ ]]** c = $\top$ "number of parameters mismatch"

LITERAL:

L **[[N]]** = *Exprval integer* (*N* **[[N]]**)

L **[[Nil]]** = *Exprval list nil*