Linear Logic

Lecture 23: Mini-ML with References in linearTwelf
April 9, 1998

1. Representation of Mini-ML with References
2. Representation of Evaluation
3. Representation of Typing Rules
4. Examples

Syntax of Mini-ML

```
exp    : type.  %name exp    E
instr  : type.  %name instr  I
cont   : type.  %name cont   K
cell   : type.  %name cell   C
item   : type.  %name item   It
store  : type.  %name store  S
answer : type.  %name answer A
```

Syntax of Mini-ML – Expressions

```
z       : exp.
s       : exp -> exp.
case    : exp -> exp -> (exp -> exp) -> exp.
lam     : (exp -> exp) -> exp.
app     : exp -> exp -> exp.
fix     : (exp -> exp) -> exp.

rf      : cell -> exp.              %prefix 80 rf
ref     : exp -> exp.
!       : exp -> exp.
noop    : exp.
;       : exp -> exp -> exp.        %infix left  90  ;
is      : exp -> exp -> exp.        %infix right 100 is
```

Syntax of Mini-ML – Instructions

```
ev : exp -> instr.
return : exp -> instr.
case*  : exp -> exp -> (exp -> exp) -> instr.
app*   : exp -> exp -> instr.

ref*   : exp -> instr.
deref* : exp -> instr.
is*1   : exp -> exp -> instr.      %infix right 100 is*1
is*2   : exp -> exp -> instr.      %infix right 100 is*2
```

Syntax of Mini-ML – Continuation, Store, Answers

```
init : cont.
-    : cont -> (exp -> instr) -> cont.  %infix left 80 -


estore : store.
and    : store -> item -> store.        %infix left 70 and
==      : cell -> exp -> item.          %infix none 60 ==


close : store -> exp -> answer.
new   : (cell -> answer) -> answer.
```

Evaluation: State of the Store

```
contains: cell -> exp -> type.   %name contains Cn

collect: store -> type.          %name collect Col

col_empty : collect estore.
col_item  : collect (S and (C == V))
             o- contains C V
             o- collect S.
```

Evaluation Relation

```
eval : cont -> instr -> answer -> type.   %name eval E

eval_z        : eval K (ev z) A
                 o- eval K (return z) A.
eval_s        : eval K (ev (s E)) A
                 o- eval (K - [x:exp] return (s x)) (ev E) A.
eval_case     : eval K (ev (case E E1 E2)) A
                 o- eval (K - [x:exp] case* x E1 E2) (ev E) A.
eval_lam      : eval K (ev (lam E)) A
                 o- eval K (return (lam E)) A.
eval_app      : eval K (ev (app E1 E2)) A
                 o- eval (K - [x:exp] app* x E2) (ev E1) A.
eval_fix      : eval K (ev (fix E)) A
                 o- eval K (ev (E (fix E))) A.
```

```
eval_cell     : eval K (ev (rf C)) A
                 o- eval K (return (rf C)) A.
eval_ref      : eval K (ev (ref E)) A
                 o- eval (K - [x:exp] ref* x) (ev E) A.
eval_deref    : eval K (ev (! E)) A
                 o- eval (K - [x:exp] deref* x) (ev E) A.
eval_noop     : eval K (ev noop) A
                 o- eval K (return noop) A.
eval_seq      : eval K (ev (E1 ; E2)) A
                 o- eval (K - [x:exp] ev E2) (ev E1) A.
eval_assign   : eval K (ev (E1 is E2)) A
                 o- eval (K - [x:exp] x is*1 E2) (ev E1) A.
```

Evaluation Relation

```
eval_init     : eval init (return V) (close S V)
               o- collect S.
eval_cont     : eval (K - I) (return V) A
               o- eval K (I V) A.
```

Evaluation Relation

```
eval_case*1    : eval K (case* z E1 E2) A
                  o- eval K (ev E1) A.
eval_case*2    : eval K (case* (s V) E1 E2) A
                  o- eval K (ev (E2 V)) A.
eval_app*      : eval K (app* (lam E1) E2) A
                  o- eval (K - [x:exp] ev (E1 x)) (ev E2) A.
eval_ref*      : eval K (ref* V) (new A)
                  o- ({c:cell} contains c V
                      -o eval K (return (rf c)) (A c)).
eval_deref*    : eval K (deref* (rf C)) A
                  o- read C V
                  &  eval K (return V) A.
eval_assign*1 : eval K ((rf C) is*1 E) A
                  o- eval (K - [x:exp] (rf C) is*2 x) (ev E) A.
eval_assign*2 : eval K ((rf C) is*2 V) A
                  o- contains C V'
                  o- (contains C V -o eval K (return noop) A).
```

Typing Judgments

```
tpc : cell -> tp -> type.          %name tpc Tc
tpe : exp -> tp -> type.           %name tpe Te
tpi : instr -> tp -> type.         %name tpi Ti
tpK : cont -> tp -> tp -> type.    %name tpK TK
tpS : store -> type.               %name tpS TS
tpa : answer -> tp -> type.        %name tpa Ta
```

Typing Expressions

```
tpe_z       : tpe z nat.
tpe_s       : tpe (s E) nat
               <- tpe E nat.
tpe_case    : tpe (case E E1 E2) T
               <- tpe E nat
               <- tpe E1 T
               <- ({x:exp} tpe x nat -> tpe (E2 x) T).
tpe_pair    : tpe (pair E1 E2) (T1 cross T2)
               <- tpe E1 T1
               <- tpe E2 T2.
tpe_fst     : tpe (fst E) T1
               <- tpe E (T1 cross T2).
tpe_snd     : tpe (snd E) T2
               <- tpe E (T1 cross T2).
```

Typing Expressions

```
tpe_lam      : tpe (lam E) (T1 arrow T2)
                <- ({x:exp} tpe x T1 -> tpe (E x) T2).
tpe_app      : tpe (app E1 E2) T1
                <- tpe E1 (T2 arrow T1)
                <- tpe E2 T2.
tpe_letval   : tpe (letval E1 E2) T2
                <- tpe E1 T1
                <- ({x:exp} tpe x T1 -> tpe (E2 x) T2).
tpe_letname : tpe (letname E1 E2) T
                <- tpe (E2 E1) T.
tpe_fix      : tpe (fix E) T
                <- ({x:exp} tpe x T -> tpe (E x) T).
```

Typing Expressions

```
tpe_cell    : tpe (rf C) (T tref)
               <- tpc C T.
tpe_ref     : tpe (ref E) (T tref)
               <- tpe E T.
tpe_deref   : tpe (! E) T
               <- tpe E (T tref).
tpe_noop    : tpe noop cmd.
tpe_seq     : tpe (E1 ; E2) T2
               <- tpe E1 T1
               <- tpe E2 T2.
tpe_assign  : tpe (E1 is E2) cmd
               <- tpe E1 (T tref)
               <- tpe E2 T.
```

Typing Instructions

```
tpi_ev       : tpi (ev E) T
                <- tpe E T.
tpi_return   : tpi (return V) T
                <- tpe V T.
tpi_case*    : tpi (case* V E1 E2) T
                <- tpe V nat
                <- tpe E1 T
                <- ({x:exp} tpe x nat -> tpe (E2 x) T).
tpi_pair*    : tpi (pair* V E) (T1 cross T2)
                <- tpe V T1
                <- tpe E T2.
tpi_fst*     : tpi (fst* V) T1
                <- tpe V (T1 cross T2).
tpi_snd*     : tpi (snd* V) T2
                <- tpe V (T1 cross T2).
```

Typing Instructions

```
tpi_app*     : tpi (app* V E) T1
                <- tpe V (T2 arrow T1)
                <- tpe E T2.
tpi_ref*     : tpi (ref* V) (T tref)
                <- tpe V T.
tpi_deref*   : tpi (deref* V) T
                <- tpe V (T tref).
tpi_assign*1: tpi (V is*1 E) cmd
                <- tpe V (T tref)
                <- tpe E T.
tpi_assign*2: tpi (V1 is*2 V2) cmd
                <- tpe V1 (T tref)
                <- tpe V2 T.
```

Typing Continuations/Store/Answers

```
tpK_init : tpK init T T.
tpK_lam  : tpK (K - I) T1 T2
            <- ({x:exp} tpe x T1 -> tpi (I x) T)
            <- tpK K T T2.

tpS_empty : tpS estore.
tpS_item  : tpS (S and (C == V))
             <- tpS S
             <- tpc C T
             <- tpe V T.

tpa_close : tpa (close S V) T
             <- tpS S
             <- tpe V T.
tpa_new   : tpa (new A) T
             <- ({c: cell} tpc c T' -> tpa (A c) T).
```

Examples: Typing and Evaluation of "z"

```
?- tpe z X.

Solving...
X = nat.
%P = tpe_z.
More? ;
No more solutions

?- eval init (ev z) V.

Solving...
V = close estore z.
%P = eval_z ^ (eval_init ^ col_empty).
More? ;
No more solutions
```

Examples: Typing and Evaluation of "app (lam x. s s x) z"

```
?- tpe (app (lam ([x] (s (s x)))) z) X.
X = nat.
%P = tpe_app tpe_z (tpe_lam
        ([x:exp] [Te1:tpe x nat] tpe_s (tpe_s Te1))).

?- eval init (ev (app (lam ([x] (s (s x)))) z)) V.
V = close estore (s (s z)).

%P = eval_app ^
      (eval_lam ^
          (eval_cont ^
              (eval_app* ^
                  (eval_z ^
    (eval_cont ^
        (eval_s ^
            (eval_s ^
                (eval_z ^
                    (eval_cont ^
                        (eval_cont ^ (eval_init ^ col_empty))))))))))).
```

Examples: Typing and Evaluation of “ref z”

```
?- tpe (ref z) X.
X = nat tref.
%P = tpe_ref tpe_z.

?- eval init (ev (ref z)) V.
V = new ([C1:cell] close (estore and (C1 == z)) (rf C1)).
%P =
   eval_ref ^
      (eval_z ^
          (eval_cont ^
              (eval_ref* ^
                  ([c:cell] [Cn1^contains c z]
                      eval_init ^ (col_item ^ col_empty ^ Cn1))))).
```

Examples:

```
let val c = ref z
in (c:=c+1;!c)
end
```

```
?- tpe (letval (ref z) ([c] (c is (s z)) ; ! c))  X.
X = nat.
%P =
   tpe_letval
      ([x:exp] [Te1:tpe x (nat tref)]
         tpe_seq (tpe_deref Te1) (tpe_assign (tpe_s tpe_z) Te1))
      (tpe_ref tpe_z).

?- eval init (ev (letval (ref z) ([c] (c is (s z)) ; ! c)))  V.
V = new ([C1:cell] close (estore and (C1 == s z)) (s z)).
%P = ...
```

Examples: Functional version of twice

```
twice n =  app (fix twice. lam x
                case x of z => z
                | s y => s (s (app twice y))) n
```

```
?- tpe (app (fix ([twice] lam ([x]
     case x z ([y] (s (s (app twice y))))))) (s (s z))) X.
X = nat.
%P =
   tpe_app (tpe_s (tpe_s tpe_z))
      (tpe_fix ....)
```

```
?- eval init (ev (app (fix ([twice] lam ([x]
     case x z ([y] (s (s (app twice y))))))) (s (s z)))) V.
V = close estore (s (s (s (s z)))).
%P =
   eval_app ^
      (eval_fix ^ ...)
```

Examples: Imperative version of twice

```
twice n = (g:=n;
           r:=0;
           fix twice. case !g of z => () | s y => (r := (s s !r); twice ());
           !r)
```

```
?- tpe (letval (ref (s (s z))) ([g]
      letval (ref z) ([r]
        (fix ([twice]  case (! g) noop ([y]
           (g is y ; r is (s (s (! r))) ; twice)))) ; (! r)))) X.

X = nat.
%P =
   tpe_letval
      ([x:exp] [Te1:tpe x (nat tref)] ...)
```

Examples: Imperative version of twice

```
twice n = (g:=n;
           r:=0;
           fix twice. case !g of z => () | s y => (r := (s s !r); twice ());
           !r)
```

```
?- eval init (ev (letval (ref (s (s z))) ([g]
        letval (ref z) ([r]
          (fix ([twice]  case (! g) noop ([y]
               (g is y ; r is (s (s (! r))) ; twice)))) ;
             (! r))))) V.
V = new
        ([C1:cell]
             new
                 ([C2:cell]
                      close (estore and (C1 == z)
                               and (C2 == s (s (s (s z)))))
                          (s (s (s (s z)))))).
%P = ...
```

Linear Logic

Lecture 23: Classical Linear Logic in linearTwelf
April 9, 1998

1. Representation of Formulas
2. Representation of Rules

Formulas

```
i : type.  % individuals
o : type.  % formulas

times  : o -> o -> o.  %infix right 11 times     % MULTIPLICATIVE
one    : o.

and    : o -> o -> o.  %infix right 11 and       % ADDITIVE
top    : o.

perp   : o -> o.                                 % INVOLUTION

forall : (i -> o) -> o.                          % QUANTIFIER

!      : o -> o.                                 % EXPONENTIALS
?      : o -> o.
```

Judgments

```
# : type.            % Token (for derivations)
neg!: o -> type.  % Modal Hypotheses (far left)
neg : o -> type.  % Hypotheses (left)
pos : o -> type.  % Conclusions (right)
pos?: o -> type.  % Modal Conclusions (far right)
```

Rules Axioms/Multiplicatives

```
axiom : (neg A -o pos A -o #).                  %% Axioms

timesr : (pos A -o #)                           %% Multiplicative
          -o (pos B -o #)
          -o (pos (A times B) -o #).

timesl : (neg A -o neg B -o #)
          -o (neg (A times B) -o #).

oner : (pos one -o #).

onel : #
        -o (neg one -o #).
```

```
andr : ((pos A -o #) & (pos B -o #))        %% Additives
        -o (pos (A and B) -o #).

andl1 : (neg A -o #)
         -o (neg (A and B) -o #).

andl2 : (neg B -o #)
         -o (neg (A and B) -o #).

topr : <T> -o (pos (top) -o #).

% no topl
```

Rules Involution/Quantifiers

```
perpr : (neg A -o #)                          %% Involution
         -o (pos (perp A) -o #).

perpl : (pos A -o #)
         -o (neg (perp A) -o #).

forallr : ({a:i} pos (A a) -o #)              %% Quantifier
           -o (pos (forall A) -o #).

foralll : {T:i} (neg (A T) -o #)
           -o (neg (forall A) -o #).
```

```
!r : (pos A -o #)                                   %% Exponentials
      -> (pos (! A) -o #).

!l : (neg! A -> #)
      -o (neg (! A) -o #).

!d : (neg A -o #)
      -o (neg! A -> #).

?r : (pos? A -> #)
      -o (pos (? A) -o #).

?l : (neg A -o #)
      -> (neg (? A) -o #).

?d : (pos A -o #)
      -o (pos? A -> #).
```

```
cut : (pos A -o #)                                  %% Cuts
       -o (neg A -o #)
       -o #.

cut! : (pos A -o #)
        -> (neg! A -> #)
        -o #.

cut? : (pos? A -> #)
        -o (neg A -o #)
        -> #.
```