Computation and Deduction

Lecture 12: Continuations
February 20, 1997

1. Correctness of Continuation Machine
2. Contextual Semantics
3. Contextual and Continuation Semantics

Mini-ML Expressions and Values I

```
exp  : type.  %name exp E
val  : type.  %name val V

z    : exp.
s    : exp -> exp.
case : exp -> exp -> (val -> exp) -> exp.
pair : exp -> exp -> exp.
fst  : exp -> exp.
snd  : exp -> exp.
lam  : (val -> exp) -> exp.
app  : exp -> exp -> exp.
letv : exp -> (val -> exp) -> exp.
letn : exp -> (exp -> exp) -> exp.
fix  : (exp -> exp) -> exp.

vl   : val -> exp.
```

Mini-ML Expressions and Values II

```
z*    : val.
s*    : val -> val.
pair* : val -> val -> val.
lam*  : (val -> exp) -> val.
```

Evaluation with Values

```
% Functions
ev_lam  : eval (lam E) (lam* E).
ev_app  : eval (app E1 E2) V
            <- eval E1 (lam* E1')
            <- eval E2 V2
            <- eval (E1' V2) V.

% Definitions
ev_letv : eval (letv E1 E2) V
            <- eval E1 V1
            <- eval (E2 V1) V.
ev_letn : eval (letn E1 E2) V
            <- eval (E2 E1) V.

% Values
ev_vl : eval (vl V) V.
```

A Continuation Machine I

```
% Machine Instructions
inst : type.  %name inst I

ev : exp -> inst.
return : val -> inst.

case1 : val -> exp -> (val -> exp) -> inst.
pair1 : val -> exp -> inst.
fst1 : val -> inst.
snd1 : val -> inst.
app1 : val -> exp -> inst.
app2 : val -> val -> inst.
let1 : val -> (val -> exp) -> inst.
```

A Continuation Machine II

```
% Continuations
cont : type.  %name cont K

init : cont.
;    : cont -> (val -> inst) -> cont.
%infix left 8 ;

% Continuation Machine States
state : type.  %name state S

# : cont -> inst -> state.
answer : val -> state.
%infix none 7 #
```

Single Step Transitions I

```
% Functions
st_lam : K # (ev (lam E)) => K # (return (lam* E)).
st_app : K # (ev (app E1 E2))
           => (K ; [x1:val] app1 x1 E2) # (ev E1).
st_app1 : K # (app1 V1 E2)
           => (K ; [x2:val] app2 V1 x2) # (ev E2).
st_app2 : K # (app2 (lam* E1') V2) => K # (ev (E1' V2)).

% Definitions
st_letv : K # (ev (letv E1 E2))
           => (K ; [x1:val] ev (E2 x1)) # (ev E1).
st_letn : K # (ev (letn E1 E2)) => K # (ev (E2 E1)).

% Values
st_vl : K # (ev (vl V)) => K # (return V).
```

Single Step Transitions II

```
% Return Instructions
st_return : (K ; C) # (return V) => K # (C V).
st_init : (init) # (return V) => (answer V).
```

Multi-Step Computations

```
%%% Multi-Step Computation
=>* : state -> state -> type.                %name =>* C
%infix none 5 =>*

stop : S =>* S.
<< : S =>* S''
      <- S => S'
      <- S' =>* S''.
%infix left 5 <<
%{
?- C : (init) # (ev (app (lam [x] (vl x)) z)) =>* (answer V).

V = z*,
C =
   stop << st_init << st_vl << st_app2 << st_return << st_z
      << st_app1 << st_return << st_lam << st_app.
}%
```

Soundness of Continuation Machine I

```
%{ Main Lemma:
   If   C :: K # (ev E) =>* (answer W)
   then for some V
        D :: eval E V
   and  C' :: K # (return V) =>* (answer W)
   where C' is a subderivation of C.

Proof: By complete induction on the structure of C.

Note that the fact that C' is a subderivation of C is not
represented in the implementation below.  This could be added
as a higher-level judgment on transformations.
}%

csd : K # (ev E) =>* (answer W)
       -> eval E V
       -> K # (return V) =>* (answer W)
       -> type.
```

Soundness of Continuation Machine II

```
% Functions
csd_lam : csd (C' << st_lam) (ev_lam) C'.

csd_app : csd (C1 << st_app) (ev_app D3 D2 D1) C'
           <- csd C1 D1 (C2 << st_app1 << st_return)
           <- csd C2 D2 (C3 << st_app2 << st_return)
           <- csd C3 D3 C'.
```

Completeness of Continuation Machine I

```
%{ Main Lemma
   If   D :: eval E V
   and  C' :: K # (return V) =>* (answer W)
   then C :: K # (ev E) =>* (answer W).

Proof: by induction on the structure of D.  C' is the
"accumulator" argument for the resulting computation.
}%

ccp : eval E V
       -> K # (return V) =>* (answer W)
       -> K # (ev E) =>* (answer W)
       -> type.
```

Completeness of Continuation Machine II

```
% Functions
ccp_lam : ccp (ev_lam) C' (C' << st_lam).

ccp_app : ccp (ev_app D3 D2 D1) C' (C1 << st_app)
           <- ccp D3 C' C3
           <- ccp D2 (C3 << st_app2 << st_return) C2
           <- ccp D1 (C2 << st_app1 << st_return) C1.

%{ from soundness:
csd_app : csd (C1 << st_app) (ev_app D3 D2 D1) C'
           <- csd C1 D1 (C2 << st_app1 << st_return)
           <- csd C2 D2 (C3 << st_app2 << st_return)
           <- csd C3 D3 C'.
}%
```

One-Step Reduction

```
==> : exp -> exp -> type.
%infix none 8 ==>

red_z      : z ==> (vl z*).
red_s      : s (vl V) ==> vl (s* V).
red_case_z : case (vl z*) E2 E3 ==> E2.
red_case_s : case (vl (s* V1')) E2 E3 ==> E3 V1'.

red_pair   : pair (vl V1) (vl V2) ==> vl (pair* V1 V2).
red_fst    : fst (vl (pair* V1 V2)) ==> (vl V1).
red_snd    : snd (vl (pair* V2 V2)) ==> (vl V2).
```

Redices

```
red_lam    : lam E1' ==> vl (lam* E1').
red_app    : app (vl (lam* E1')) (vl V2) ==> E1' V2.

red_letv   : letv (vl V1) E2 ==> E2 V1.
red_letn   : letn E1 E2 ==> E2 E1.

red_fix    : fix E ==> E (fix E).
% no red_vl rule

%%% Redices
redex : exp -> type.

rdx : redex E
       <- E ==> E'.
```

Splitting Expressions I

```
split : (exp -> exp) -> exp -> (exp -> exp) -> exp -> type.

%{ split C E C' E'
   Evaluation context C and expression E are given,
   C' and E' are constructed, E' is a redex.
   Invariant: (C E) == (C' E')
}%
% Redices
sp_redex : split C E C E
            <- redex E.

% Natural Numbers
% no sp_z
sp_s : split C (s E1) C' E'
        <- split ([h:exp] C (s h)) E1 C' E'.

sp_case : split C (case E1 E2 E3) C' E'
           <- split ([h:exp] C (case h E2 E3)) E1 C' E'.
```

Splitting Expressions II

```
% Pairs
sp_pair2 : split C (pair (vl V1) E2) C' E'
            <- split ([h:exp] C (pair (vl V1) h)) E2 C' E'.
sp_pair1 : split C (pair E1 E2) C' E'
            <- split ([h:exp] C (pair h E2)) E1 C' E'.
sp_fst   : split C (fst E) C' E'
            <- split ([h:exp] C (fst h)) E C' E'.
sp_snd   : split C (snd E) C' E'
            <- split ([h:exp] C (snd h)) E C' E'.

% Functions
% no sp_lam
sp_app2 : split C (app (vl V1) E2) C' E'
           <- split ([h:exp] C (app (vl V1) h)) E2 C' E'.
sp_app1 : split C (app E1 E2) C' E'
           <- split ([h:exp] C (app h E2)) E1 C' E'.
```

Splitting Expressions III

```
% Definitions
sp_letv : split C (letv E1 E2) C' E'
           <- split ([h:exp] C (letv h E2)) E1 C' E'.
% no sp_letn
% Recursion
% no sp_fix
% Values
% no sp_vl

%%% Top-Level Splitting

split_exp : exp -> (exp -> exp) -> exp -> type.

spe : split_exp E C E'
       <- split ([h:exp] h) E C E'.
```

Contextual Evaluation

```
%%% One-Step Contextual Evaluation
one_step : exp -> exp -> type.

ostp : one_step E (C R')
        <- split_exp E C R
        <- R ==> R'.

%%% Full Contextual Evaluation
xeval : exp -> val -> type.

xev_vl : xeval (vl V) V.
xev_step : xeval E V
            <- one_step E E'
            <- xeval E' V.
```