Parameter Passing

Handout #36
26 April 2006

Call-by-Value (CBV)

- Argument expressions are evaluated and results are passed to a function.
- Example (HOFL substitution model):

```
((fun (x y) (* x x)) (+ 1 2) (* 3 4))
⇒ ((fun (x y) (* x x)) 3 12) ; First eval args
⇒ (* 3 3) ; Then substitute values
⇒ 9 ; Then continue evaluation
```

  Each argument expression is evaluated exactly once regardless of whether or not it is used in the body of the function.
- Alternative characterization: environments map names to (cells of) values.

Call-by-Value: HOFL/HOILEC Implementation

```
(* val eval: exp -> valu Env.env -> valu *)
let rec eval exp env =
  match exp with
    ⋮
   | Var name ->
       (match Env.lookup name env with
          Some v -> v
        | None -> raise (EvalError("Unbound variable: " ^ name)))
  | Abs(fml,body) -> Fun(fml,body,env) (* make a closure *)
  | App(rator,rand) -> (* force left-to-right evaluation *)
      let fcn = (eval rator env) in
      let arg = (eval rand env) in apply fcn arg
    ⋮

and apply fcn arg =
  match fcn with
    Fun(fml,body,senv) -> eval body (Env.bind fml arg senv)
  | _ -> raise (EvalError ("Non-function rator in application: "
                           ^ (valuToString fcn)))
```

Call-by-Value: HOILIC Implementation

```
(* val eval: exp -> valu ref Env.env -> valu *)
let rec eval exp env =
  match exp with ...
   | Var name ->
       (match Env.lookup name env with
          Some cell -> ! cell
        | None -> raise (EvalError("Unbound variable: " ^ name)))
   | Assign(name,rhs) ->
       (match Env.lookup name env with
          Some cell -> let oldValu = (! cell)
                       and newValu = eval rhs env in
                       let _ = cell := newValu in oldValu
         | None -> raise (EvalError("Unbound variable: " ^ name)))
  | Abs(fml,body) -> Fun(fml,body,env) (* make a closure *)
  | App(rator,rand) -> (* force left-to-right evaluation *)
      let fcn = (eval rator env) in
      let arg = (eval rand env) in apply fcn arg

and apply fcn arg =
  match fcn with
    Fun(fml,body,senv) -> eval body (Env.bind fml (ref arg) senv)
  | _ -> raise (EvalError ...)
```

Call-by-Value Example

Describe the evaluation of `(testpp)` in CBV HOILEC:

```
(def (test a b c)
  (seq (println "enter")
       (bind result (+ c (* b b))
         (seq (println "exit")
              result))))

(def (testpp)
  (test (println (+ 1 2))
        (println (+ 3 4))
        (println (+ 5 6))))
```

Call-by-Value Languages

- Most modern languages are call-by-value (e.g. OCaml, Scheme, Java, C, Pascal value parameters).
- OCaml is like the HOILEC implementation: variables are bound directly to values, not cells holding values.
- Java, C, Pascal, Scheme are like the HOILIC implementation: each variable is bound to an implicit cell automatically dereferenced at each variable reference.

Call-by-Name (CBN)

- Unevaluated argument expressions are passed to a function. An argument expression is re-evaluated every time the corresponding parameter is used in the body. It is never evaluated if it is never used.
- Example (HOFL substitution model):

```
((fun (x y) (* x x)) (+ 1 2) (* 3 4))
⇒ (* (+ 1 2) (+ 1 2)) ; Substitute unevaled arg exps
⇒ (* 3 3)  ; Continue evaluation
⇒ 9
```

- Each argument expression is evaluated the number of times it is used in the function body. Better than call-by-value for arguments never used, but worse for arguments used more than once.
- Alternative characterization: environments map names to (cells of) thunks (i.e., unmemoized promises).
- ALGOL-60 was a call-by-name language.

Call-by-Name: HOFL/HOILEC Implementation

```
(* val eval: exp -> (unit -> valu) Env.env -> valu *)
let rec eval exp env =
  match exp with ...
  | Var name ->
     (match Env.lookup name env with
        Some thunk -> thunk() (* dethunk *)
      | None -> raise (EvalError("Unbound variable: " ^ name)))
  | Seq(exp1,exp2) -> (* In CBN, seq is a kernel construct *)
      let _ = eval exp1 env in eval exp2 env
  | Abs(fml,body) -> Fun(fml,body,env) (* make a closure *)
  | App(rator,rand) -> (* force left-to-right evaluation *)
      let fcn = (eval rator env) in
      let arg = (fun () -> eval rand env) in
      apply fcn arg

  and apply fcn arg =
    match fcn with
      Fun(fml,body,env) -> eval body (Env.bind fml arg env)
    | _ -> raise (EvalError ("Non-function rator in application: "
                             ^ (valuToString fcn)))
```

Call-by-Name Example

Describe the evaluation of `(testpp)` in CBN HOILEC:

```
(def (test a b c)
  (seq (println "enter")
       (bind result (+ c (* b b))
         (seq (println "exit")
              result))))

(def (testpp)
  (test (println (+ 1 2))
        (println (+ 3 4))
        (println (+ 5 6))))
```

Call-by-Lazy (CBL) a.k.a. Call-by-Need

- Unevaluated argument expressions are passed to a function. An argument is evaluated only the first time its parameter is used in the body, and that value is used thereafter. It is never evaluated if the parameter is never used.
- Example (HOFL/HOILEC substitution model):

```
((fun (x y) (* x x)) (+ 1 2) (* 3 4))
; Substitute (but share) unevaluated argument expressi
⇒ (* .  .) ⇒ (* . .) ⇒ 9
      \ /        \ /
   (+ 1 2)        3
```

- Each argument expression is evaluated once if it is used in the function body, but zero times if it is never used. This is the best case scenario!
- Alternative characterization: environments map names to (cells of) memoized promises.
- The Haskell language uses call-by-lazy.

Call-by-Lazy: HOFL/HOILEC Implementation

```
(* val eval: exp -> valu Promise.promise Env.env -> valu *)
let rec eval exp env =
  match exp with ...
  | Var name ->
      (match Env.lookup name env with
        Some promise -> Promise.force promise
      | None -> raise (EvalError("Unbound variable: " ^ name)))
  | Seq(exp1,exp2) -> (* In CBL, seq is a kernel construct *)
      let _ = eval exp1 env in eval exp2 env
  | Abs(fml,body) -> Fun(fml,body,env) (* make a closure *)
  | App(rator,rand) -> (* force left-to-right evaluation *)
      let fcn = (eval rator env) in
      let arg = Promise.make (fun () -> eval rand env) in
      apply fcn arg

  and apply fcn arg =
    match fcn with
      Fun(fml,body,env) -> eval body (Env.bind fml arg env)
    | _ -> raise (EvalError ("Non-function rator in application: "
                             ^ (valuToString fcn)))
```

Call-by-Lazy Example

Describe the evaluation of `(testpp)` in CBL HOILEC:

```
(def (test a b c)
  (seq (println "enter")
       (bind result (+ c (* b b))
         (seq (println "exit")
              result))))

(def (testpp)
  (test (println (+ 1 2))
        (println (+ 3 4))
        (println (+ 5 6))))
```

Relating CBV, CBN, and CBL

- In a purely functional language, evaluating expression E under call-by-name and call-by-lazy always gives the same result.
- In a purely functional language, if E evaluates to values V_V, V_N, and V_L under call-by-value, call-by-name, and call-by-lazy (respectively), then V_V, V_N, and V_L must be the same value.
- However, call-by-name/need will sometimes return values in cases where call-by-value fails to do so (because of errors or infinite loops). E.g.:

```
((fun (x y) (* x x)) (+ 1 2) (/ 3 0))
((fun (x y) (* x x)) (+ 1 2) (loop))
  ; Suppose (loop) loops infinitely
```

- In an imperative language, all bets are off. That is, for some expressions, each mechanism can return a completely different value.

Call-by-Reference (CBR)

All argument expressions are evaluated as in CBV. But we pass to a function the reference cell of any parameter that is a variable, and create a new reference cell for parameter that is not a variable.

```
;; CBR example in HOILIC
(bind a 0
  (bind inc (fun (x)
              (seq (<- x (+ x 1))
                   a))
    (list a ; In both CBV and CBR, returns 0
          (inc a) ; CBV returns 0; CBR returns 1
          (inc a))) ; CBV returns 0; CBR, returns 2
```

Call-by-Reference: HOILIC Implementation

```
(* val eval : exp -> valu ref Env.env -> valu *)
let rec eval exp env = match exp with
   App(rator,rand) ->
     let fcn = (eval rator env) in
     let arg = (leval rand env) in apply fcn arg
  | ... (* all other clauses same as for CBV *)

(* val leval : exp -> valu ref Env.env -> valu ref *)
and leval exp env = match exp with (* Evaluate "left" value *)
    | Var name -> (match Env.lookup name env with
          Some cell -> cell (* Return cell, not contents *)
        | None -> raise (EvalError("Unbound variable: " ^ name)))
    | If(tst,thn,els) -> (match eval tst env with
                          Bool true -> leval thn env
                        | Bool false -> leval els env
                        | v -> raise (EvalError ...))
    | _ -> ref (eval exp env) (* else create fresh cell for value *)

(* val apply: fcn -> valu ref -> valu *)
and apply fcn argref = match fcn with
    Fun(fml,body,env) -> eval body (Env.bind fml argref env)
  | _ -> raise (EvalError ...)
```

Call-by-Reference in Pascal

Pascal supports both call-by-value and call-by-reference. Call-by-reference parameters are distinguished with a `var` keyword in parameter declarations.

```
program ParamTest (input,output);
    var a, b: integer;
    procedure paramTest (x:integer, var y:integer);
        begin
            x := x + y;
            y := x * y;
        end;
    begin
        a := 3;
        b := 4;
        paramTest(a,b);
        writeln('a=', a); {a is still 3}
        writeln('b=', b); {b is now 28}
    end;
end.
```

A Call-by-Reference Swap in Pascal

```
program TestSwap;
    procedure swap (var x : int, var y : int);
        begin
            var temp:integer := x;
            x := y;
            y := temp;
        end;
    begin
        var a:integer := 1;
        var b:integer := 2;
        swap(a,b); {a now contains 2 and b contains 1}
        {Can also call swap on array slots:
          e.g. swap(c[i],d[j]).}
    end;
end.
```

Simulating Call-by-reference in C

C is CBV, but has pointer operations for simulating CBR.

- The *address-of operator* (&) returns location of (i.e. pointer to) a variable.
- The dereference operator (*) returns contents of a pointed-at variable.

```
void paramTest (int x, int* y)
{
    x = x + *y;
    *y = x * *y;
}

int main (int argc, char *argv[])
{
    int a = 3;
    int b = 4;
    paramTest(a,&b);
    printf("a=%d; b=%d\n", a, b);
}

% gcc -o paramtest paramtest.c
% paramtest
a=3; b=28
```

Swap Example in C

```
void printab (int x, int y) { printf("a=%d; b=%d\n", x, y);}

void swap (int* x, int* y)
{
    int temp;
    printf("x=%u; *x=%d; y=%u; *y=%d\n",x,*x,y,*y);
    temp = *x; *x = *y; *y = temp;
    printf("x=%u; *x=%d; y=%u; *y=%d\n",x,*x,y,*y);
}
int main (int argc, char *argv[])
{
    int a = 1;
    int b = 2;
    printab(a,b);
    swap(&a,&b); // Can also swap array slots: e.g. swap(&c[i], &d[j])
    printab(a,b);
}
% gcc -o swap swap.c; ./swap
a=1; b=2
x=3221223268; *x=1; y=3221223264; *y=2
x=3221223268; *x=2; y=3221223264; *y=1
a=2; b=1
```

Call-by-reference in C++

C++ supports call-by-reference parameters:

```
void swap (int &x, int &y)
{
    int temp = x;
    x = y;
    y = temp;
}

int main (int argc, char *argv[])
{
    int a = 1;
    int b = 2;
    swap(a,b); // a now contains 2 and b contains 1
    // Can also swap array slots: e.g. swap(c[i], d[j])
}
```

Simulating Call-by-reference in OCaml, Scheme, and Java

In OCaml, call-by-reference simulated by passing explicit cells (references):

```
fun  swap (x, y) =
    temp = ! x;
    x := ! y;
    x := temp

let a = ref 3
let b = ref 4
let  _ = swap(a, b)
```

The same trick works in Scheme and Java. Note that there is no way to access the implicit cells that variables are bound to in Scheme and Java, so it is impossible to write a swap function on the implicit cells. E.g., if a and b are Java variables, there is no swap function such that `swap(a,b)` swaps the values of a and b.