Parameter Passing

Handout #38

CS251 Lecture 30

April 12, 2002

Call-by-Value

In call-by-value, pass to a function the values resulting from evaluating the argument expressions.

Example (HOFL substitution model):

```
   ((abs (x y) (* x x)) (+ 1 2) (* 3 4))
-> ((abs (x y) (* x x)) 3 12) ; First evaluate 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 values (or cells containing values in an imperative language).

Call-by-Value: HOFL/HOILEC Implementation (without bindrec)

```
evalExp : Exp -> Val Ident.Env -> Val

evalExp (VarRef(name)) env =
  (case Env.lookup(name,env) of
     NONE => raise EvalError (“Unbound variable”)
   | SOME(v) => v)

evalExp (FunApp(rator,rands)) env =
  let val fcn = evalExp rator env
      val args = evalExps rands env
   in funapply fcn args
  end

funapply (ClosureVal(fmls,body,env)) argVals =
  evalExp body
          (Env.extend(fmls,argVals,env))
funapply _ env = raise EvalError “applying non-closure”
```

Call-by-Value: HOILIC Implementation

```
evalExp : Exp -> Val ref Ident.Env -> Val

evalExp (VarRef(name)) env =
  (case Env.lookup(name,env) of
     NONE => raise EvalError (“Unbound variable”)
   | SOME(ref v) => v) (* Dereference value from cell *)

evalExp (FunApp(rator,rands)) env =
  let val fcn = evalExp rator env
      val args = evalExps rands env
   in funapply fcn args
  end

funapply (ClosureVal(fmls,body,env)) argVals =
  evalExp body
          (Env.extend(fmls,
                      map ref argVals, (* Implicit cells *)
                      env))
funapply _ env = raise EvalError “applying non-closure”
```

Call-by-Value Languages

Most modern languages are call-by-value (e.g. C, Java, Scheme, ML, Pascal's value parameters).

- ML is like the HOILEC implementation: variables are bound directly to values, not cells holding values.
- C, Pascal, Java, Scheme are like the HOILIC implementation: each variable is bound to an implicit cell automatically dereferenced at each variable reference.

Parameter Passing Test

Assume that `iprint` displays an integer followed by a newline and returns the integer. Assume `sprint` works similarly for strings.

E.g. assuming left-to-right evaluation, `(+ (iprint 1) (iprint 2))` prints a 1 and 2 on the console and returns 3 (the result of the addition).

Consider the following `test` function in HOILIC:

```
(abs (a b c)
  (seq (sprint “enter”)
       (bind result (+ c (* b b))
         (seq (sprint “exit”)
              result))))
```

What does the following print under various parameter passing mechanisms?

```
(test (iprint (+ 1 2)) (iprint (+ 3 4)) (iprint (+ 5 6)))
```

Call-by-Name

In call-by-name, pass to a function the unevaluated argument expressions. The argument is re-evaluated every time it is used in the body. (It is never evaluated if it is never used.)

Example (HOFL substitution model):

```
   ((abs (x y) (* x x)) (+ 1 2) (* 3 4))

-> (* (+ 1 2) (+ 1 2)) ; Substitute unevaluated
                       ; argument expressions
-> (* 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) non-memoizing promises.

ALGOL-60 was a call-by-name language.

Call-by-Name: HOFL/HOILEC Implementation (without bindrec)

```
datatype Promise = Delay of Exp * Promise Ident.Env

evalExp : Exp -> Promise Ident.Env -> Val

evalExp (VarRef(name)) env =
  (case Env.lookup(name,env) of
     NONE => raise EvalError (“Unbound variable”)
   | SOME(Delay(exp,env)) = evalExp exp env)

evalExp (FunApp(rator,rands)) env =
  let val fcn = evalExp rator env
      val argPromises = map (fn r => Delay(r,env)) rands
   in funapply fcn argPromises
  end

funapply (ClosureVal(fmls,body,env)) argPromises =
  evalExp body
          (Env.extend(fmls, argPromises ,env))
```

Simulating Call-by-Name in Call-by-Value

Can simulate call-by-name in a call-by-value language by wrapping a thunk around every argument and dethunking every variable reference.

For example, here is the call-by-name parameter test expressed in Scheme:

```
(define test
  (lambda (a b c)
    (begin (print “enter”)
           (let ((result (lambda () (+ (c) (* (b) (b))))))
             (begin (display “exit”)
                    (result))))))
```

```
(test (lambda () (print (+ 1 2)))
      (lambda () (print (+ 3 4)))
      (lambda () (print (+ 5 6))))
```

Call-by-Need (Lazy Evaluation)

In call-by-need, pass to a function the unevaluated argument expressions. The argument is evaluated only the *first time* it is used in the body, and that value is used thereafter. (It is never evaluated if it is never used.)

Example (HOFL)

```
   ((abs (x y) (* x x)) (+ 1 2) (* 3 4))
  ; Substitute (but share) unevaluated argument expressions
```

```
-> (*  .   . )    ->  (* .   .)   ->  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: envs map names to (cells of) memoized promises.

The Haskell language uses call-by-need.

Call-by-Need: HOFL/HOILEC Implementation (without bindrec)

```
datatype Promise = Delay of Exp * Promise Ident.Env * Val option ref

evalExp : Exp -> Promise Ident.Env -> Val

evalExp (VarRef(name)) env =
  case Env.lookup(name,env) of
    NONE => raise EvalError ("Unbound variable")
  | SOME(Delay(exp,env,memo)) =
     case !memo of
       NONE => let val v = evalExp exp env
                in (memo := SOME(v); v) end
     | SOME(v) => v

evalExp (FunApp(rator,rands)) env =
  let val fcn = evalExp rator env
      val delayedArgs = map (fn r => Delay(r,env,ref NONE))
                            rands
   in funapply fcn delayedArgs
  end
```

Simulating Call-by-Need in Call-by-Value

Can simulate call-by-need in a call-by-value language by wrapping a promise around every argument and forcing every variable reference.

For example, here is the call-by-need parameter test expressed in Scheme:

```
(define test
  (lambda (a b c)
    (begin (print "enter")
           (let ((result (delay (+ (force c)
                                   (* (force b)
                                      (force b))))))
             (begin (display "exit")
                    (force result))))))

(test (delay (print (+ 1 2)))
      (delay (print (+ 3 4)))
      (delay (print (+ 5 6))))
```

Relationship between by-value, by-name, by-need

In a purely functional language, evaluating expression *E* under call-by-name and call-by-need always gives the same result.

In a purely functional language, if *E* evaluates to values *V1*, *V2*, and *V3* under call-by-value, call-by-name, and call-by-need (respectively), then *V1*, *V2*, and *V3* 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.:

```
((lambda (x y) (* x x)) (+ 1 2) (/ 3 0))
((lambda (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

In call-by-reference, pass to a function the reference cell of any parameter that is a variable (create a new reference cell for parameters that are not variables).

```
;; HOILIC example
;; In HOILIC, “<-” updates the implicit cell of a variable
;; (like set! in Scheme)
(bindseq ((a 0)
          (inc (abs (x) (seq (<- x (+ x 1)) a))))
  (prepend a ; In both CBV and CBR, returns 0
    (prepend (inc a) ; CBV returns 0; CBR, returns 1
      (prepend (inc a) ; CBV returns 0; CBR, returns 2
        (empty)))))
```

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 testRef
  procedure p (x : int, var y : int);
    begin
      x := x * 2;
      y := y + x
    end;
  begin
    var a : int := 3;
    var b : int := 4;
    p(a,b);
    {a is still 3, but b is now 10}
  end
end.
```

Call-by-Reference: swap

You can use call-by-reference to write a swap function. E.g., in Pascal:

```
program testSwap
  procedure swap (var x : int, var y : int);
    begin
      var temp : int := x;
      x := y;
      y := temp
    end;
  begin
    var a := 3;
    var b := 4;
    swap(a,b);
    {Could also call swap on array slots. E.g.
     swap(c[i], d[j])}
  end
end.
```

Simulating call-by-reference in C

Although C is a call-by-value language, it has “features” that allows simulating call-by-reference.

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

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

int a = 3;
int b = 4;
swap(&a, &b);
/* Can also use on arrays. E.g.: swap(&c[i], &d[j]) */
```

Call-by-reference in C++

C++ is an object-oriented extension to C that supports a call-by-reference parameter passing mode.

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

int a = 3;
int b = 4;
swap(a, b);
/* Can also use on arrays. E.g.: swap(c[i], d[j]) */
```

Simulating call-by-reference in ML, Scheme, and Java

In ML, the effect of call-by-reference can be achieved by passing *explicit* cells (references)

```
fun swap (x, y) =
  let val temp = (^ x)
      val _ = x := (^ y)
      val _ = y := temp
   in ()
  end

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

The same trick can be pulled 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 Scheme variables, there is no `swap` function such that `(swap a b)` swaps the values of `a` and `b`.

Compound Data

Parameter passing issues are more complex in the context of compound data structures, which can be allocated either on the **stack** (Execution Land) or in the **heap** (Object Land).

In ML, Scheme, and Java:

- all compound values are allocated in the heap.
- All compound values are "small" pointers to heap-allocated objects that are automatically dereferenced.
- Inaccessible objects are automatically reclaimed by *garbage collection*.

In C and Pascal:

- Can choose between allocating compound data on stack (the default) vs. heap (C's `malloc`, Pascal's `new`).
- Stack-allocated compound values are "big" values in structured variables.
- All pointers must be dereferenced explicitly (using C's *, Pascal's ↑).
- Programmer must manually reclaim inaccessible objects (C's `free`, Pascal's `dispose`.)

Points in C

```
typedef struct P {int x; int y;} point;

point scaledCopy (int s, point p)
  { point q; q.x = s * p.x; q.y = s * p.y; return q; }

void scale1 (int s, point p)
  { p.x = s * p.x; p.y = s * p.y; }

void scale2 (int s, point* p)
 { (*p).x = s * (*p).x; (*p).y = s * (*p).y; }

void printPoint (point p)
  { printf("x=%d;y=%d\n", p.x, p.y); }

int main () {
  { point a,b; a.x = 1; a.y = 2;
    b = scaledCopy(3,a); printPoint(a); printPoint(b);
    scale1(4,a); scale2(5,&b); printPoint(a); printPoint(b); }
```

Integer Lists in C

```
typedef struct IL {int head; struct IL *tail;} intlist;

int sumlist (intlist* lst)
{ if (lst == NULL) return 0;
  else return (*lst).head + sumlist((*lst).tail);}

intlist* fromTo (int lo, int hi)
{ intlist* result;
  if (lo > hi) return NULL;
  else {result  = (intlist*) malloc(sizeof(intlist));
        (*result).head = lo;
        (*result).tail = fromTo(lo + 1, hi);
        return result;}}

int main () {
{ printf("sumlist(fromTo(1,10))=%d\n",
         sumlist(fromTo(1,10)));}
```