Subprograms II:
Parameter Passing

CAS CS 320: Principles of Programming Languages

Thursday, April 11, 2024

Administrivia

- Project 1 (i.e. Homework 9) posted Friday, Apr 5, due Monday, Apr 15.
- Final exam on Wednesday, May 8, 3:00-5:00 pm in STO 50.

Parameter Passing =

Evaluation Strategy

(https://en.wikipedia.org/wiki/Evaluation_strategy)

Parameter Passing

(slides composed by Andrew Appel of Princeton University)

Call-by-value Evaluation

[redacted] OCaml is *call-by-value (CBV)*

Also called *strict* or *eager*.

Left-to-right CBV evaluation of a function application e1 e2:

1) e1 is evaluated to a value v1, which should be a function (fun x -> e)
2) e2 is evaluated to a value v2
3) evaluation continues by substituting v1 for x in the body of the expression e

```
    (fun x -> x + x) (2+3)
--> (fun x -> x + x) 5
--> 5 + 5
--> 10
```

Note that OCaml doesn't specify whether it is left-to-right CBV or right-to-left CBV.

Right-to-left CBV evaluation of a function application:

1) e2 is evaluated to a value v2
2) e1 is evaluated to a value v1, which should be a function (fun x -> e)
3) evaluation continues by substituting v1 for x in the body of the expression e

Call-by-value Evaluation

Notice that the following expression evaluates the same way regardless of whether we use left-to-right or right-to-left CBV

left-to-right CBV:

```
    (fun x -> x + x) (2+3)
--> (fun x -> x + x) 5
--> 5 + 5
--> 10
```

right-to-left CBV:

```
    (fun x -> x + x) (2+3)
--> (fun x -> x + x) 5
--> 5 + 5
--> 10
```

Call-by-value Evaluation

The following expression is evaluated in a slightly different order under left-to-right or right-to-left CBV:

left-to-right CBV:

```
    (fun x -> fun y -> x + y) 2) (3+5)
--> (fun y -> 2 + y) (3+5)
--> (fun y -> 2 + y) 8
--> 2 + 8
--> 10
```

right-to-left CBV:

```
     (fun x -> fun y -> x + y) 2) (3+5)
-->  (fun x -> fun y -> x + y) 2) 8
-->  (fun y -> 2 + y) 8
--> 2 + 8
--> 10
```

But notice that they compute the same value in the end.

Left-to-right and right-to-left CBV evaluation in pure languages (with effects) always gives the same answer.

Specifying Evaluation Orders

There are many more ways that one might evaluate a functional program! (We saw one: lazy evaluation)

If we want to specify how a language evaluates precisely, we can use an *operational semantics*.

We typically specify operational semantics using inference rules.

Recall:

$$\frac{\text{premiss1} \qquad \text{premiss2} \qquad \ldots \qquad \text{premissn}}{\text{conclusion}}$$

"if premiss1 and premiss2 ... and premiss3 are all valid then the conclusion is valid"

valid means "can be proven by finitely many other inference rules"

λ-calculus

The pure λ-calculus is a language that contains nothing but variables, functions, and function application:

```
x         -- just a variable
λx.e      -- a function with parameter x and body e (i.e., fun x -> e)
e1 e2     -- one expression applied to another (function application)
```

The only lambda calculus *values* are functions (λx.e).

When you see the letter v in what follows, assume I am referring to a value. When you see the letter e, assume I am referring to a general expression.

λ-calculus operational semantics

CBV evaluation rules:

$$\frac{}{(\lambda\, x\, .\;\; e)\; v \;\;\mapsto\;\; e[v/x]} \quad (\beta\text{-reduction})$$

$$\frac{e1 \;\;\mapsto\;\; e1'}{e1\; e2 \;\;\mapsto\;\; e1'\; e2}$$

$$\frac{e2 \;\;\mapsto\;\; e2'}{e1\; e2 \;\;\mapsto\;\; e1\; e2'}$$

Examples:

```
    (λx. x x) (λy.y)
--> (λy.y) (λy.y)
```

```
    ( (λx. x x) (λy.y) )  ( (λx. x x) (λy.y) )
--> ( (λy.y) (λy.y)     )  ( (λx. x x) (λy.y) )
```

```
    ( (λx. x x) (λy.y) )  ( (λx. x x) (λy.y) )
--> ( (λx. x x) (λy.y) )  ( (λy.y) (λy.y)     )
```

λ-calculus operational semantics

Left-to-right CBV evaluation rules:

$$\frac{}{(\lambda\, x\, .\ e)\ v \quad \mapsto \quad e[v/x]} \quad (\beta\text{-reduction})$$

$$\frac{e1 \quad \mapsto \quad e1'}{e1\ e2 \quad \mapsto \quad e1'\ e2}$$

$$\frac{e2 \quad \mapsto \quad e2'}{v\ e2 \quad \mapsto \quad v\ e2'}$$

Examples:

```
    (λx. x x) (λy.y)
--> (λy.y) (λy.y)
```

```
    ( (λx. x x) (λy.y) )  ( (λx. x x) (λy.y) )
--> ( (λy.y) (λy.y)     )  ( (λx. x x) (λy.y) )
```

Doesn't apply because green is not a value:

```
    ( (λx. x x) (λy.y) )  ( (λx. x x) (λy.y) )
--> ( (λx. x x) (λy.y) )  ( (λy.y) (λy.y)     )
```

λ-calculus operational semantics

Call-by-Name (CBN) evaluation rules:

$$\frac{}{(\lambda\, x\, .\ e)\ e2 \ \mapsto\ e[e2/x]}\ (\beta\text{-reduction})$$

$$\frac{e1 \ \mapsto\ e1'}{e1\ e2 \ \mapsto\ e1'\ e2}$$

$$\cancel{\frac{e2 \ \mapsto\ e2'}{e1\ e2' \ \mapsto\ e1\ e2'}}$$

Examples:

(λx. x x) ((λy.y) (λy.y))
--> ((λy.y) (λy.y)) ((λy.y) (λy.y))

((λx. x x) (λy.y)) ((λx. x x) (λy.y))
--> ((λy.y) (λy.y)) ((λx. x x) (λy.y))

Don't evaluate expressions until you have to.
Just substitute them in for parameters of functions

Pragmatic CBN Examples

```
    (fun x -> fun y -> x + y + y) 2) (3+5)
--> (fun y -> 2 + y + y) (3+5)
--> 2 + (3+5) + (3+5)
--> 2 + 8 + (3+5)
--> 10 + (3+5)
--> 10 + 8
--> 18
```

I decided to evaluate operators left-to-right

	Printed So Far
`    (fun x -> x; x ) (print_string "hi")`	
`--> (print_string "hi"); (print_string "hi")`	
`--> (); print_string "hi"`	hi
`--> print_string "hi"`	hi
`--> ()`	hihi

Non-terminating Computations

Consider the following computation:

$$(\lambda x.\ x\ x)\ (\lambda y. y\ y)$$

What does it evaluate to using left-to-right CBV evaluation?

$$(\lambda y.\ y\ y)\ (\lambda y. y\ y)$$

That is the same thing (modulo variable renaming)!

That thing is not a value ... we can keep computing ... forever

We also get the same result if we use right-to-left CBV or CBN!

Do we always get the same answer?

Consider the following computation:

(λx. λy.y) (loop)

where loop is (λy. y y) (λy.y y)

What does it evaluate to using CBV evaluation in 1 step?

(λx. λy.y) (loop)

where loop is (λy. y y) (λy.y y)

What does it evaluate to using CBN evaluation in 1 step?

λy.y

Sometimes call-by-name terminates when call-by-value doesn't!

Is CBN always better than CBV?

Consider the following computation:

```
(λx. x x) (big)

where big is (((λy. y) (λy.y)) (λy. y)) (λy.y)
```

CBV evaluates "big" once.

CBN evaluates "big" twice:

```
(λx. x x) (big)
--> (big) (big)
```

Any time a parameter is used more than once in a function body, CBN is going to repeat evaluation of the argument. Not good!

Parameter Passing

(slides composed by Professor Louis Steinberg of Rutgers University)

Parameter Passing Methods

Procedural abstraction

- **Parameter passing methods**
  - **pass by value**
  - **pass by result**
  - **pass by value-result**
  - **pass by reference**
    - **aliasing**
  - **pass by name**
- **Procedures/functions as arguments**

Pass by Value

```
{   c: array [1..10] of integer;
    m,n : integer;
    procedure r (k,j : integer)
    begin
        k := k+1;
        j := j+2;
    end r;
…
    m := 5;
    n  := 3;
    r(m,n);
    write m,n;
}
```

By Value:

k	j
~~5~~	~~3~~
6	5

Output:
5 3

Pass by Value

- **Advantages**
  - **Argument protected from changes in callee**
- **Disadvantages**
  - **Copying of values takes execution time and space, especially for aggregate values**

Pass by Result

```
{   c: array [1..10] of integer;
    m,n : integer;
    procedure r (k,j : integer)
    begin
        k := k+1;
        j := j+2;
    end r;
…
    m := 5;
    n  := 3;
    r(m,n);
    write m,n;
}
```

Error in procedure r: can't use parameters which are uninitialized!

Pass by Value-Result

```
{   c: array [1..10] of integer;
    m,n : integer;
    procedure r (k,j : integer)
    begin
            k := k+1;
            j := j+2;
    end r;
…
    m := 5;
    n  := 3;
    r(m,n);
    write m,n;
}
```

By Value-Result

k	j
~~5~~	~~3~~
6	5

Output:
6 5

Pass by Value-Result

```
{   c: array [1..10] of integer;
    m,n : integer;
    procedure r (k,j : integer)
    begin
            k := k+1;
            j := j+2;
    end r;
  /* set c[m] = m */
    m := 2;
    r(m, c[m]);
    write c[1], c[2], …, c[10];
}
```

k	j
~~2~~	~~2~~
3	4

What element of c has its value changed? c[2]? c[3]?

Pass by Reference

```
{   c: array [1..10] of integer;
    m,n : integer;
    procedure r (k,j : integer)
    begin
            k := k+1;
            j := j+2;
    end r;
…
    m := 5;
    n  := 3;
    r(m,n);
    write m,n;
}
```

k	j
--> m	-->n

m	n
~~5~~	~~3~~
6	5

Value update happens in storage of the caller while callee is executing

Comparisons

- **Value-result**
  - **Has all advantages and disadvantages of value and result together**
- **Reference**
  - **Advantage: is more efficient than copying**
  - **Disadvantage: can redefine constants**

    r(0, X) **will redefine the constant zero in old Fortran'66 compilers**
  - **Leads to aliasing: when there are two or more different names for the same storage location**
    - **Side effects not visible from code itself**

Aliasing: by Reference

```
{  y: integer;
   procedure p(x: integer)
   {  x := x + 1;
      x := x + y;
   }
...
   y := 2;
   p(y);
   write y;
}
```

x
-->y

during the call, x and y are the same location!

y
~~2~~
~~3~~
6

output: 6

No Aliasing: Value-Result

```
{ y: integer;
  procedure p(x: integer)
  { x := x + 1;
    x := x + y;
  }
…
  y := 2;
  p(y);
  write y;
}
```

x: ~~2~~ ~~3~~ 5

y: ~~2~~ 5

output: 5

Another Aliasing Example

```
{   j, k, m :integer;
    procedure q( a, b: integer)
    {   b := 3;
        m := m *a;
    }
...
s1:  q(m, k);
…
s2: q(j, j);
…
}
```

global-formal aliases:
<m,a> <k,b> associations during call S1;

formal-formal aliases:
<a,b> during call S2;

Pass by Reference

- **Disadvantage: if an error occurs, harder to trace values since some side-effected values are in environment of the caller**
- **What happens when someone uses an expression argument for a by reference parameter?**
  - **(2*x)??**

Pass by Name

```
{   c: array [1..10] of integer;
    m,n : integer;
    procedure r (k,j : integer)
    begin
          k := k+1;
          j := j+2;
    end r;
/* set c[n]  to n */
    m := 2;
    r(m,c[m]);
    write m,n;
}
```

m:= m+1

c[m] := c[m] + 2

m	c[ ]
~~2~~	1 2 ~~3~~ 4 5 6 7 8 9 10
3	1 2 5 4 5 6 7 8 9 10

Parameter Passing

(more examples of parameter-passing, courtesy of *CSE 505: Concepts of Programming Languages* at the University of Washington)

Example 1: illustrates call by value, value-result, reference

```
begin
integer n;
procedure p(k: integer);
    begin
    n := n+1;
    k := k+4;
    print(n);
    end;
n := 0;
p(n);
print(n);
end;
```

Note that when using call by reference, `n` and `k` are aliased.

Output:

```
call by value:         [redacted]
call by value-result:
call by reference:
```

Example 1: illustrates call by value, value-result, reference

```
begin
integer n;
procedure p(k: integer);
    begin
    n := n+1;
    k := k+4;
    print(n);
    end;
n := 0;
p(n);
print(n);
end;
```

Note that when using call by reference, `n` and `k` are aliased.

Output:

```
call by value:        1 1
call by value-result: 1 4
call by reference:    5 5
```

Example 2: Call by value and call by name

```
begin
integer n;
procedure p(k: integer);
    begin
    print(k);
    n := n+1;
    print(k);
    end;
n := 0;
p(n+10);
end;
```

Output:

```
call by value:
call by name:
```

[redacted]

Example 2: Call by value and call by name

```
begin
integer n;
procedure p(k: integer);
    begin
    print(k);
    n := n+1;
    print(k);
    end;
n := 0;
p(n+10);
end;
```

Output:

```
call by value:    10 10
call by name:     10 11
```

Example 3: Call by value and call by name (with evaluation errors)

```
begin
integer n;
procedure p(k: integer);
    begin
    print(n);
    end;
n := 5;
p(n/0);
end;
```

Output:

```
call by value:    [redacted] .
call by name:
```

Example 3: Call by value and call by name (with evaluation errors)

```
begin
integer n;
procedure p(k: integer);
    begin
    print(n);
    end;
n := 5;
p(n/0);
end;
```

Output:

```
call by value:    divide by zero error
call by name:     5
```

Example 4: Non-local references

```
  procedure clam(n: integer);
  begin

    procedure squid;
    begin
      print("in procedure squid -- n="); print(n);
    end;

    if n<10 then clam(n+1) else squid;

  end;

clam(1);
```

Output:

```
in procedure squid -- [redacted]
```

Example 4: Non-local references

```
  procedure clam(n: integer);
  begin

    procedure squid;
    begin
      print("in procedure squid -- n="); print(n);
    end;

    if n<10 then clam(n+1) else squid;

  end;

clam(1);
```

Output:

```
in procedure squid -- n=10
```

(THIS PAGE INTENTIONALLY LEFT BLANK)