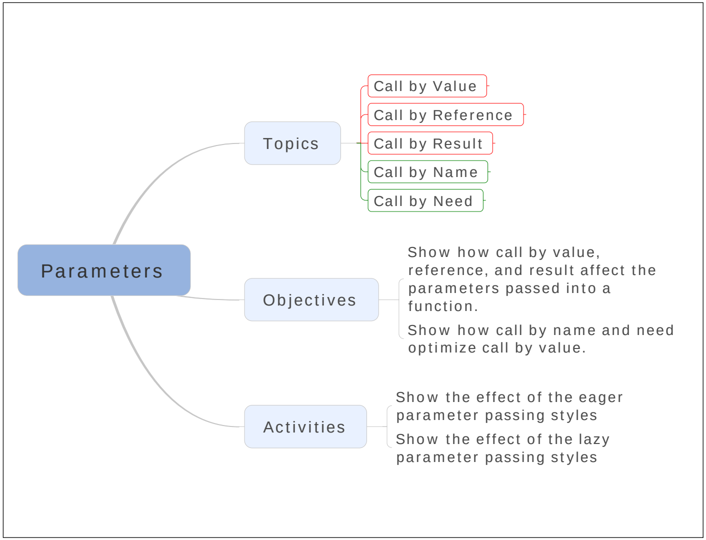
Parameter Passing Styles

Dr. Mattox Beckman

Illinois Institute of Technology
Department of Computer Science

Outline


Parameters
Topics
Call by Value
Call by Reference
Call by Result
Call by Name
Call by Need
Objectives
Show how call by value, reference, and result affect the parameters passed into a function.
Show how call by name and need optimize call by value.
Activities
Show the effect of the eager parameter passing styles
Show the effect of the lazy parameter passing styles

Objectives

You should be able to...

The function call is one of the most fundamental elements of programming. The meaning of a function call is greatly affected by the choice of parameter passing style.

- Understand five kinds of parameter passing:
    1. Call By Value
    2. Call By Reference
    3. Call By Name
    4. Call By Result
    5. Call By Value-Result

Running Example

We will use the following code to illustrate the concepts:

```
let foo x y z =
  x := z * z * y;  (* let's pretend that this *)
  y := 5;          (* is legal *)
  x + y

let main () =
  let a = 10 in
  let b = 20 in
    foo a b (a+b)
```

Call By Value

- Parameters are evaluated before the function call takes place.
- The function receives a copy of the parameters.
  - Changes made to variables in the function are not visible outside.
- Advantages: speed
- Disadvantage: instability

```
# let pi1 a b = a
val pi1 : 'a -> 'b -> 'a = <fun>
# let rec foo () =  pi1 5 (foo ());;
val foo : unit -> int = <fun>
# foo ();;
Stack overflow during evaluation (looping recursion?).
```

Result of CBV

```
let foo x y z =
  x := z * z * y;
  y := 5;
  x + y
```

```
let main () =
  let a = 10 in
  let b = 20 in
    foo a b (a+b)
```

- `a` is copied into `x`.
- `b` is copied into `y`.
- `a+b` is evaluated to 30, the 30 is copied into `z`.
- `x` is assigned 30 * 30 * 20.
- `y` is assigned 5.
- upon return, `a` and `b` have their original values.
- This is used by C, C++, OCaml, ...“most languages”.

Call By Reference

- Parameters are evaluated before the function call takes place.
- The function receives a copy of the parameters.
- Variables are passed as pointers.
  - Changes made to variables in the function are visible outside.
- Advantages: speed, saves some memory, side effects are possible when you want them.
- Disadvantage: side effects are possible when you don't want them.

Result of Call by Reference

```
let foo x y z =
  x := z * z * y;
  y := 5;
  x + y

let main () =
  let a = 10 in
  let b = 20 in
    foo a b (a+b)
```

- a and x share the same memory.
- b and y share the same memory.
- a+b is evaluated to 30, the 30 is copied into z.
- x and a are assigned 30 * 30 * 20.

- y and b are assigned 5.
- upon return, a and b have new values.
- Used by C, C++, OCaml optionally; Java by default.

Example

```
int inc(int i) {
  return ++i;
}

int main() {
  int i = 10;
  cout << inc(i) << ” ” << i << endl;
}
```

What will be the output of this code?

Example

```
int inc(int &i) {
  return ++i;
}

int main() {
  int i = 10;
  cout << inc(i) << ” ” << i << endl;
}
```

What will be the output of this code?

Call By Result

- Parameters are updated before the function call *returns*.
- Often combined with call by value. Call by result, call by value, and call by value-result are “subclasses” of call-by-copy. What changes is when the copy occurs.
  - Changes made to variables in the function are visible outside—in fact, that’s the whole point.
- Advantages: you can return multiple values from a single function
- Disadvantages: variables can be clobbered inadvertently.

Result of Call By Result

```
let a = 10
let b = 20

let foo x y z =
  x := z * z * y;
  y := 5;
  a + b

let main () =
    foo a b (a+b)
```

- `a` is copied into `x`.
- `b` is copied into `y`.
- `a+b` is evaluated to 30, the 30 is copied into `z`.
- `x` is assigned 30 * 30 * 20.

- `y` is assigned 5.
- `a + b` will evaluate to 30
- upon return, `x` is copied into `a`, and `y` is copied into `b`.
- This is used by Prolog. (Sort of...)

Call By Name

- Parameters are evaluated after the function call is made.
- The parameters are substituted into the function body.
- Changes made to variables in the function *are* visible outside.
- Advantages: stability
- Disadvantage: inefficiency — computations can be duplicated

```
# let pi1 a b = a;;
val pi1 : 'a -> 'b -> 'a = <fun>
# let rec foo () =  pi1 5 (foo ());;
val foo : unit -> int = <fun>
# foo ();;
val - : int = 5
```

Result of Call By Name

```
let foo x y z =
  x * x + y * y

let main () =
    foo (10+10) (20+20)
        (main ())
```

- `x` is replaced by `(10+10)`.
- `y` is replaced by `(20+20)`.
- `z` is replaced by `(main ())`.
- The call to `main` via `z` never happens.
- The + operation happens five times.

- This was used by Algol. Also used by some “term rewriting” systems.

Call By Need

- Parameters are encapsulated into a *thunk*.
- The thunks are passed into the function.
- The first time a thunk is executed, the value is cached.
- Remaining executions use the cached value.
- Advantages: stability
- Disadvantage: efficient, but time sensitive.

```
# let pi1 a b = a;;
val pi1 : 'a -> 'b -> 'a = <fun>
# let rec foo () =  pi1 5 (foo ());;
val foo : unit -> int = <fun>
# foo ();;
val - : int = 5
```

Result of Call By Need

```
let foo x y z =
  x * x + y * y

let main () =
    foo (10+10) (20+20)
        (main ())
```

- x is replaced by a pointer to (10+10).
- y is replaced by a pointer to (20+20).
- z is replaced by a pointer to (main ()).

- The call to main via z never happens.
- The + operation happens only once for each variable.
- This is used by Haskell. Also known as *lazy evaluation*.
- Not compatible with assignment.

Activity

Do the Parameter Passing Style activity.