CSC 347 - Concepts of Programming Languages

Argument Passing

Instructor: James Riely

Learning Objectives

❓ How should arguments to functions be passed in?

- Interpret call-by-value and call-by-reference function calls

Argument Passing

❓ What are `x`, `y`, `"hello"`, and `10` below:

```
1 def f (x:String, y:Int) = x * y
2 f ("hello", 10)
```

- `x`, `y`: *formal parameters* (or *parameters*)
- `"hello"`, `10`: *actual parameters* (or *arguments*)

Argument Passing

- What does a call to `f` print?

```
1 def g (y:Int) : Int =
2   y = y + 1
3 end g
4
5 def f () =
6   var x = 1
7   g (x)
8   println (x)
9 end f
```

- Prints `1` because Scala uses call-by-value

Call-By-Value vs. Call-By-Reference

Call-by-Value

- Most PLs use *call-by-value* (CBV) by default
- To run `g (e)`
    i. evaluate `e` to a value `v`
    ii. pass a *copy* of `v` to `g`
    iii. callee changes to copy of `v` are not visible to caller

Call-by-Reference

- Some PLs use *call-by-reference* (CBR)
- To run `g (e)`
    i. evaluate `e` to an l-value `r`
    ii. pass the address `r` to `g`
    iii. callee changes via `r` are visible to caller

Example

```
1 def g(y:Int) : Int =
2   y = y + 1
3 end g
4
5 def f() =
6   var x = 1
7   g(x)
8   println(x)
9 end f
```

Call-By-Value

- Prints `1` in a CBV PL, i.e., `x=1` after call to `g`

Call-By-Reference

- Prints `2` in a CBR PL, i.e., `x=2` after call to `g`

Formal Semantics of Function Calls

- ❓ How do we specify what a function does?
- Have: global store ξ for variables and their values
- Need another store ϕ: maps function names to argument names and function bodies
- Need a local store ρ for arguments and local variables

Call-by-value

$$\frac{\left.\begin{array}{c}\langle \mathtt{e}_1,\xi_0,\phi,\rho_0\rangle \Downarrow \langle v_1,\xi_1,\phi_1,\rho_1\rangle \\ \vdots \\ \langle \mathtt{e}_n,\xi_{n-1},\phi,\rho_{n-1}\rangle \Downarrow \langle v_n,\xi_n,\phi_n,\rho_n\rangle\end{array}\right\}\text{evaluate arguments } \mathtt{e_n} \qquad \phi(\mathtt{f})=(\mathtt{x_1},\ldots,\mathtt{x_n},\mathtt{b}) \quad \text{look up function} \qquad \langle \mathtt{b},\xi_n,\phi,\{\mathtt{x_1}\mapsto v_1,\ldots,\mathtt{x_n}\mapsto v_n\}\rangle \Downarrow \langle v,\xi',\phi',\rho'\rangle \quad \text{evaluate function body } \mathtt{b}}{\langle \mathtt{f(e_1,\ldots,e_n)},\xi_0,\phi,\rho_0\rangle \Downarrow \langle v,\xi',\phi,\rho_n\rangle}\ (\text{FunCBV})$$

Call-by-reference (copy-in, copy-out)

$$\frac{\left.\begin{array}{c}\langle \mathtt{e}_1,\xi_0,\phi,\rho_0\rangle \Downarrow \langle v_1,\xi_1,\phi_1,\rho_1\rangle \\ \vdots \\ \langle \mathtt{e}_n,\xi_{n-1},\phi,\rho_{n-1}\rangle \Downarrow \langle v_n,\xi_n,\phi_n,\rho_n\rangle\end{array}\right\}\text{evaluate arguments } \mathtt{e_n} \qquad \phi(\mathtt{f})=(\mathtt{x_1},\ldots,\mathtt{x_n},\mathtt{b}) \quad \text{look up function} \qquad \langle \mathtt{b},\xi_n,\phi,\{\mathtt{x_1}\mapsto v_1,\ldots,\mathtt{x_n}\mapsto v_n\}\rangle \Downarrow \langle v,\xi',\phi',\rho'\rangle \quad \text{evaluate function body } \mathtt{b}}{\langle \mathtt{f(e_1,\ldots,e_n)},\xi_0,\phi,\rho_0\rangle \Downarrow \langle v,\xi',\phi,\rho_n \cup \rho'_{\downarrow\{x_1,\ldots,x_n\}}\rangle}\ (\text{FunCBR})$$

CBR and Temporaries

❓ Can temporary values be passed as l-values?

- `g(x+1)` is not obviously legitimate in CBR
- Some languages reject it, some allow it
- Perl allows it

Call-By-Reference: Perl

- Perl uses CBR

```perl
sub g {
    $_[0] = $_[0] + 1;
}

sub f {
    my $x = 1;
    g ($x);
    print ("x = $x\n");
}

f ();
```

```
$ perl ./cbr.pl
x = 2
```

Call-By-Reference: Perl

- Perl allows temporaries!

```perl
sub g {
    $_[0] = $_[0] + 1;
}

sub f {
    my $x = 1;
    g ($x + 1);
    print ("x = $x\n");
}

f ();
```

```
$ perl ./cbr.pl
x = 1
```

Call-By-Reference: Perl

- Simulate CBV by creating copies explicitly

```
 1 sub g {
 2     my ($y) = @_;
 3     $y = $y + 1;
 4 }
 5
 6 sub f {
 7     my $x = 1;
 8     g ($x);
 9     print ("x = $x\n");
10 }
11
12 f ();
13 # x=1
```

Call-By-Value: C

- Simulate CBR in C by explicitly passing, receiving, accessing a pointer

```c
void g (int *p) {
  *p = *p + 1;
}

int main () {
  int x = 1;
  g (&x);
  printf ("x = %d\n", x);
  return 0;
}
// x=2
```

Call-By-Reference: C++

- C++ reference type `int&` (unlike `int*`, creates references implicitly)

```
1 void g(int& y) {
2   y = y + 1;
3 }
4
5 int main() {
6   int x = 1;
7   g(x);
8   // x==2
9   cout << "x = " << x << endl;
10  return 0;
11 }
```

❓ What makes this design error-prone?

Call-By-Reference: C++

❗ In C++, at caller CBR is indistinguishable from CBV

```
1 void g(int  y) { // vs. void g(int& y)
2   y = y + 1;
3 }
4
5 int main() {
6   int x = 1;
7   g(x);
8   // x==1 vs. x==2 when void g(int& y)
9   cout << "x = " << x << endl;
10  return 0;
11 }
```

- Small change in function signature, large effect at caller

Call-By-Reference: C#

- C# has reference parameters `ref int`

```
 1 class Test {
 2   static void g (ref int y) {
 3     y = y + 1;
 4   }
 5
 6   static void Main () {
 7     int x = 1;
 8     g (ref x);
 9     // x == 2
10     Console.WriteLine("{0}", x);
11   }
12 }
```

- Unlike `int&` must also be used by caller

Call-By-Reference: C++

- Passing a non-lvalue

```
1  void g(int& y) {
2    y = y + 1;
3  }
4
5  int main() {
6    int x = 1;
7    g(x + 1);
8    cout << "x = " << x << endl;
9    return 0;
10 }
```

```
1 $ g++ -o reference reference.cpp
2 reference.cpp: In function ‘int main()’:
3 referencs.cpp:11:8: error: invalid initialization of non-const
4   reference of type ‘int&’ from an rvalue of type ‘int’
5    g (x + 1);
6         ^
7 references.cpp:5:6: error: in passing argument 1 of ‘void g(int&)’
8  void g (int& y) {
9       ^
```

Call-By-Value: Java

- Java has only a restricted form of *pointers*, called *references*
  - must point to heap-allocated objects
  - cannot point to stack-allocated data
  - cannot point to primitive types
- Java references cannot be forged
  - not from integers via casting
  - not from other references via pointer arithmetic
- Objects only accessed via references
  - unlike C++
  - Java has no address-of `&` operator

Call-By-Value: Java

- Simulate CBR In Java: heap-allocated object with field of intended argument type
- Pass a reference to the object instance **by value**

```
1  class IntRef { int n; }
2
3  public class Ref {
4    static void g(IntRef r) { r.n = r.n + 1; }
5
6    public static void main(String[] args) {
7      IntRef s = new IntRef (); s.n = 1;
8      g(s);
9      // s.n == 2
10     System.out.println (s.n);
11   }
12 }
```

Summary

Call-by-value

- Pass copies of r-values as arguments
- Changes not visible to caller
- Strict evaluation

Call-by-reference

- Pass l-values as arguments
- Changes visible to caller
- Strict evaluation