CSC 347 - Concepts of Programming Languages

Closures

Instructor: James Riely

Learning Objectives

❓ What if functions have free variables?

- Identify when a function has free variables
- Identify nested (local) methods and functions
- Describe the effect of closures: how free variables are bound to variables of the environment

Local Variables and Functions

- Local variables: limit their scope
- ❓ Does this work for functions too?

```c
int fact (int n) {
  int loop(int n, int result) {
    if (n <= 1) {
      return result;
    } else {
      return loop(n - 1, n * result);
    }
  }
  return loop(n, 1);
}
```

```
$ gcc -c nested-fact.c
$ gcc -pedantic -c nested-fact.c
function.c: In function ‘fact’:
function.c:2:3: warning: ISO C forbids nested functions [-pedantic]
```

- Allowed by GCC, but not C standard

```scala
def fact(n:Int) : Int =
  def loop(n:Int, result:Int) : Int =
    if n<=1 then result
    else loop(n-1, n*result)
  end loop
  loop(n, 1)
end fact
```

- Defined in Scala

Local Functions: Challenge

```c
typedef void (*funcptr) (int);

// f returns pointer to local function g
funcptr f (int x) {
  void g (int y) {
    printf ("x = %d, y = %d\n", x, y);
  }
  return &g;
}

int main (void) {
  funcptr h1 = f (10);
  (*h) (2); // call g through h
  f (20);
  (*h) (3); // call g through h
}
```

Where is `x`?

```
x = 10, y = 2 # (*h)(2): unsafe call h
x = 20, y = 3 # (*h)(3): unsafe call h
```

Local Functions: Java and Scala

- Closures: local functions in Scala and Java store environment on function definition

Scala

```
def f(x:Int) : Int=>Unit =
  def g(y:Int) = println(s"x=$x, y=$y")
  g // return function g, which accesses x
end f
val h1 = f(10)
h1(2)           // prints x=10, y=2
val h2 = f(20)
h1(3)           // prints x=10, y=3
h2(3)           // prints x=20, y=3
```

- Java

```
1  import java.util.function.Function;
2
3  public static Function<Integer,Void> f(int x) {
4    Function<Integer,Void> g = y -> {
5      System.out.format ("x=%d, y=%d%n", x, y);
6      return null;
7    }
8    return g;
9  }
10 public static void main (String[] args) {
11   Function<Integer,Void> h1 = f(10);
12   h1.accept(2);           // prints x=10, y=2
13   Function<Integer,Void> h2 = f(20);
14   h1.accept(3);           // prints x=10, y=3
15   h2.accept(3);           // prints x=20, y=3
16 }
```

Closures: Copy or Share

```
1 def outer(x:A) : B=>C =
2   def inner(y:B) : C =
3     // use x and y
4     // free: x
5     // bound: y
6   end inner
7   inner // return inner
8 end outer
```

Closure contains

- Code for `inner`
- Copy of `x`

Closures: Copy or Share

```
 1 def outer(x:A) : B=>C =
 2   var u:A = x
 3   def inner(y:B) : C =
 4     // use x, u, and y:
 5     // free: x, u
 6     // bound: y
 7   end inner
 8   u = u + 1
 9   inner // return inner
10 end outer
```

Closure contains

- Code for `inner`
- Copy of `x`
- What to do with `u`?
  - `inner` sees updated `u`?
    - Scala
  - require `u` to be immutable?
    - Java

Closures: Scala

- `inner` has free `x`
- Needs closure

```
def outer(x:Int) : Boolean=>Int =
  def inner(y:Boolean) : Int =
    x + (if y then 0 else 1)
  end inner
  inner // return inner
end outer
```

- Object-oriented implementation in Java

```
public static Function1<Boolean, Integer> outer(int x) {
  return new Closure(x);
}

public final class Closure
    extends AbstractFunction1<Boolean, Integer> {

  private final int x;

  public final Integer apply(Boolean y) {
    return x + (y ? 0 : 1);
  }

  public Closure(int x) { this.x = x; }
}
```

Closures: Scala

- `inner` has free `x`, `u`
- Needs closure

```
1 def outer (x:Int) : Boolean=>Int =
2   var u:Int = x
3   def inner (y:Boolean) : Int =
4     x + u + (if y then 0 else 1)
5   end inner
6   u = u+1
7   inner // return inner
8 end outer
```

- Object-oriented impl. stores mutable `u` on heap

```
 1 public static Function1<Boolean, Integer> outer(int x) {
 2   IntRef u = new IntRef(x);
 3   var c = new Closure(x, u);
 4   u.elem = u.elem+1;
 5   return c;
 6 }
 7
 8 public final class Closure
 9     extends AbstractFunction1<Boolean, Integer> {
10   private final int x;
11   private final IntRef u;
12   public final Integer apply(Boolean y) {
13     return x + u.elem + (y ? 0 : 1);
14   }
15   public Closure(int x, IntRef u) {
16     this.x = x;
17     this.u = u;
18   }
19 }
```

Example: Use Closures

Data encapsulation in a "container" for data and functions

Object-oriented programming: classes

```java
public class Incrementor {
  private int i;
  public Incrementor(int i) {
    this.i = i;
  }
  public int increment(int x) {
    return x+i;
  }
}
// use object
Incrementor inc = new Incrementor(2);
inc.increment(4); // returns 6
inc.increment(5); // returns 7
```

❓ Functional programming

```scala
def incrementor(i:Int) : Int=>Int =
  (x:Int) => x+i
end incrementor



// use closure
val inc = incrementor(2)
inc(4) // returns 6
inc(5) // returns 7
```

- Useful for languages without strong visibility modifiers (Python, JavaScript)

Example: Interpret Closures

- Closure for method

```
1 val f: ()=>Int =
2   var x = -1
3   def g() =
4     x = x + 1
5     x
6   end g
7   g
```

- Closure for function

```
1 val f: ()=>Int =
2   var x = -1
3   () => {
4     x = x + 1
5     x
6   }
```

- Initializes:
  - `x` to `-1` when initializing variable `f`
- Returns:
  - incremented `x` on every call `f()`

```
1 scala> f()
2 res0: Int = 0
3 scala> f()
4 res1: Int = 1
```

Example: Interpret Closures

- Closure for method

```
def g(y: Int) : ()=>Int =
  var z = y
  def h() =
    z = z + 1
    z
  end h
  h
```

- Closure for function

```
val g: Int=>()=>Int =
  y => {
    var z = y
    () => {
      z = z + 1
      z
    }
  }
```

- `g(i)` returns:
  - function `()=>Int`, its own `z` set to `i`

```
scala> val h1=g(10)
h1: () => Int = $$Lambda$1098/39661414@54d8c20d
scala> val h2=g(20)
h2: () => Int = $$Lambda$1098/39661414@5bc7e78e
```

- `h1()` and `h2()` return:
  - their own incremented `z`

```
scala> h1()
res3: Int = 11
scala> h1()
res4: Int = 12
scala> h2()
res5: Int = 21
scala> h1()
res6: Int = 13
```

Example: Use Closures for Partial Functions

Implement a function that prints a greeting

Scala

```scala
val greet = greeter("Hello")
println(greet("Alice"))
println(greet("Bob"))
```

- Return a function for printing that accesses the outer greeting argument

```scala
def greeter(greeting: String) : String=>String =
  (name: String) => s"$greeting $name!"
```

JavaScript

- Event handlers with preserved state

```javascript
const button1 = document.getElementById("button1");
const button2 = document.getElementById("button2");

const greet = createGreeter("Hello");
button1.addEventListener("click", greet("Alice"));
button2.addEventListener("click", greet("Bob"));
```

```javascript
function createGreeter(greeting) {
  function greet(name) {
    return function() {
      console.log(greeting + " " + name + "!");
    }
  }
  return greet;
}
```

Example: Use Closures for Stateful Functions

Implement a function that collects numbers and computes their average:

```
1 val cavg = cumulativeAvg()
2 cavg(2) // returns 2
3 cavg(4) // returns 3
4 cavg(6) // returns 4
```

- Initialize a list buffer, return a function that appends and then computes the average

```
1 def cumulativeAvg(): Int=>Int =
2   val data = scala.collection.mutable.ListBuffer.empty[Int]
3   x =>
4     data.append(x)
5     data.sum / data.length
6 end cumulativeAvg
```

Example: Use Closures for Decorators

Implement a function to cache the results of another function:

Scala

```scala
def square(x: Int) =
  println(s"$x^2")
  x*x
end square
val cachedSquare = memoize(square)
cachedSquare(5) // prints 5^2 and returns 25
cachedSquare(5) // returns 25 without printing
```

- Initialize cache, return function that consults and updates cache

```scala
def memoize[X,Y](fn: X=>Y) : X=>Y =
  import scala.collection.mutable.Map
  val cache = Map.empty[X,Y]
  x => cache.getOrElseUpdate(x, fn(x))
```

Python

```python
def memoize(function):
  cache = {}
  def closure(number):
    if number not in cache:
      cache[number] = function(number)
    return cache[number]
  return closure
```

- Use decorator

```python
@memoize
def square(x):
  print(f"{x}^2")
  return x*x
```

Example: Use Closures for Data Encapsulation

Implement a map data structure from arrays and index access; wanted use:

```
1 val (get, put) = map(10)
2 put(3, "Hello")
3 put(6, "world!")
4 println(get(3))
```

- Return a tuple of functions that access a shared array

```
1 def map(size: Int) : (Int=>String, (Int, String)=>Unit) =
2   val elems: Array[String] = Array.ofDim(size)
3
4   def get(k: Int)             = elems(k)
5   def put(k: Int, v: String) = elems(k) = v
6
7   (get, put)
8 end map
```

Example: Debouncing and Throttling

- Debouncing: delayed execution of a function (e.g., search after typing finished)
- Throttling: limit the number of executions in a time interval

```
const searchInput =
  document.getElementById("searchInput");

 // Delay by 500ms after last keystroke
const debouncedSearch = debounce(
  function(searchTerm) {
    console.log("Search for:", searchTerm);
  },
  500
);

searchInput.addEventListener(
  "keyup",
  function(event) {
    debouncedSearch(event.target.value);
  }
);
```

```
function debounce(func, delay) {
  let timeout;

  return function(...args) {
    clearTimeout(timeout);
    timeout = setTimeout(() => {
      func.apply(this, args);
    }, delay);
  };
}
```

Summary

- Closures are necessary when functions have free variables
- Closures bind free variables of a function to variables from the environment
- Closures align the lifetime of functions and their accessed environment
- Closures in Javascript