CSC 347 - Concepts of Programming Languages

Scope and Lifetime

Instructor: Stefan Mitsch

Learning Objectives

How should identifiers relate to memory locations?

- Interpret global, static, and dynamic scope
- Identify when shadowing occurs
- Identify bugs related to the difference between the scope of an identifier and the lifetime of a memory location

Scope

- *Scope* of an identifier: region of text in which it may be used

```
def f (x: Int) =
  val y = x+1
  if x>y then
    val z = y+1
    println(s"z = $x")
  end if
end f
```

- `x` and `y` are in scope after their declaration until end of method `f`
- `z` is in scope after its declaration until end of `if`-block

Occurrences of Identifiers

- *Free* occurrence has no matching binding

```
y = 5*x;    // Free occurrences of x and y
```

- *Binding* occurrence declares the identifier

```
val y : Int;    // binding occurrence of y
```

- *Bound* occurrence follows matching declaration

```
var y : Int;    // Binding occurrence of y
var x : Int;    // Binding occurrence of x

x = 6;     // Bound occurrence of x
y = 5*x;   // Bound occurrences of x and y
```

Binding and Circular Dependencies

What to do with circular dependencies?

```
char f (int x) { return x>0 ? g (x-1) : 1; }
```

```
char g (int x) { return f (x) + f (x); }
```

- Most modern languages allow any order
- C, C++ require *forward declarations*

```
char f (int x);
char g (int x);
// f and g definitions can now be in any order
```

Formal Semantics for Functions Revisited

- Global store ξ for variables and their values
- Function store ϕ: maps function names to argument names and function bodies
- Local store ρ for arguments and local variables

Call-by-value

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

? Where to look up variables?

$$\dfrac{}{\langle \mathtt{x},\xi,\phi,\rho\rangle \Downarrow \langle \xi(\mathtt{x}),\xi,\phi,\rho\rangle}\ (\text{Var})$$

! Variable lookup revisited: global scope

Global Scope: Python

- Python `global` accesses the global scope

```python
def useX():
  print (x)

def defX():
  global x
  x = 1
```

```
>>> useX()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in useX
NameError: name 'x' is not defined

>>> defX()
>>> useX()
1
```

Clearly not ideal; what other scoping rules could we define?

Local-Then-Global Scope

❓ What does this program do, using Scala *syntax*, but various different *semantics*?

```
var x: Int = 1

def g() =
  var x: Int = 2

  def f() =
    x = 3
  end f

  f(); println(s"x-g-f = $x")
  x = 2

  def h() =
    var x: Int = 4
    f(); println(s"x-h = $x")
  end h

  h(); println(s"x-g-h = $x")
end g

g(); println(s"x-m = $x")
```

- Local store, then global store
  - $$\frac{\mathtt{x} \in \rho}{\langle \mathtt{x}, \xi, \phi, \rho \rangle \Downarrow \langle \rho(\mathtt{x}), \xi, \phi, \rho \rangle} \text{(VarLocal)}$$
  - $$\frac{\mathtt{x} \notin \rho \qquad \mathtt{x} \in \xi}{\langle \mathtt{x}, \xi, \phi, \rho \rangle \Downarrow \langle \xi(\mathtt{x}), \xi, \phi, \rho \rangle} \text{(VarGlobal)}$$
  - $$\frac{\mathtt{x} \notin \rho \qquad \mathtt{x} \notin \xi}{\langle \mathtt{x}, \xi, \phi, \rho \rangle \Downarrow \langle \mathtt{error}, \xi, \phi, \rho \rangle} \text{(VarUndef)}$$
- Prints
  - `x-g-f = 2`
  - `x-g-h = 2`
  - `x-h = 4`
  - `x-m = 3`

Dynamic Scope

What does this program do, using Scala *syntax*, but various different *semantics*?

```
var x: Int = 1

def g() =
  var x: Int = 2

  def f() =
    x = 3
  end f

  f(); println(s"x-g-f = $x")
  x = 2

  def h() =
    var x: Int = 4
    f(); println(s"x-h = $x")
  end h

  h(); println(s"x-g-h = $x")
end g

g(); println(s"x-m = $x")
```

- List of local stores, then global store
  - $\dfrac{\rho_h :: \rho_t = \rho \qquad \mathbf{x} \in \rho_h}{\langle \mathbf{x}, \xi, \phi, \rho \rangle \Downarrow \langle \rho_h(\mathbf{x}), \xi, \phi, \rho \rangle}$ (VarLocal)
  - $\dfrac{\rho_h :: \rho_t = \rho \qquad \mathbf{x} \notin \rho_h}{\langle \mathbf{x}, \xi, \phi, \rho \rangle \Downarrow \langle \mathbf{x}, \xi, \phi, \rho_t \rangle}$ (VarRec)
  - $\dfrac{\rho = \emptyset \qquad \mathbf{x} \in \xi}{\langle \mathbf{x}, \xi, \phi, \rho \rangle \Downarrow \langle \xi(\mathbf{x}), \xi, \phi, \rho \rangle}$ (VarGlobal)
  - $\dfrac{\rho = \emptyset \qquad \mathbf{x} \notin \xi}{\langle \mathbf{x}, \xi, \phi, \rho \rangle \Downarrow \langle \mathtt{error}, \xi, \phi, \rho \rangle}$ (VarUndef)
- Prints
  - `x-g-f = 3`
  - `x-g-h = 2`
  - `x-h = 3`
  - `x-m = 1`

Static Scope

What does this program do, using Scala *syntax*, but various different *semantics*?

```
var x: Int = 1

def g() =
  var x: Int = 2

  def f() =
    x = 3
  end f

  f(); println(s"x-g-f = $x")
  x = 2

  def h() =
    var x: Int = 4
    f(); println(s"x-h = $x")
  end h

  h(); println(s"x-g-h = $x")
end g

g(); println(s"x-m = $x")
```

- Bind variables to store when defining functions (ρ maps variables to stores, rather than variables to values)
  - $$\frac{\mathtt{x}\in\rho \qquad \rho_h=\rho(x) \qquad \mathtt{x}\in\rho_h}{\langle \mathtt{x},\xi,\phi,\rho\rangle \Downarrow \langle \rho_h(\mathtt{x}),\xi,\phi,\rho\rangle}\ (\text{VarLocal})$$
  - $$\frac{\mathtt{x}\notin\rho \qquad \mathtt{x}\in\xi}{\langle \mathtt{x},\xi,\phi,\rho\rangle \Downarrow \langle \xi(\mathtt{x}),\xi,\phi,\rho\rangle}\ (\text{VarGlobal})$$
  - $$\frac{\mathtt{x}\notin\rho \qquad \mathtt{x}\notin\xi}{\langle \mathtt{x},\xi,\phi,\rho\rangle \Downarrow \langle \mathtt{error},\xi,\phi,\rho\rangle}\ (\text{VarUndef})$$
- Prints
  - `x-g-f = 3`
  - `x-g-h = 3`
  - `x-h = 4`
  - `x-m = 1`

Static and Dynamic Scope

Static Scope

- Compile time scoping, also known as **lexical scope**
- Identifiers are bound to the closest binding occurrence in an **enclosing block** of the program code
- Renaming any identifier to a fresh name consistent throughout its scope does not change program behavior

Dynamic Scope

- Runtime scoping
- Identifiers are bound to the binding occurrence in the **closest activation record**
- Consistent renaming may break a working program!

Static and Dynamic Scope

- Where could `z` come from?

```
...
def g (x:Int) : Int =
  var y:Int = x * 2
  z * x * y            // x and y are bound; z is free
```

- Static scope
  - Free variables are resolved (bound) at compile time
  - Look at outer blocks in source code
- Dynamic scope
  - Free variables are resolved (bound) at runtime
  - Look at the caller activation records on the stack

Static Scope: Scala

- Scala uses static scope
- Most languages do use static scope: Java, C, C++, Python (unless `global`) etc.

```scala
var x:Int = 10      // x0

def f () : Unit =
  x = 20            // x is bound statically to x0

def g () : Unit =
  var x:Int = 30.   // x1
  f ()

g ()
println (x) // prints 20
```

Dynamic Scope: Bash

- Bash uses dynamic scope

```bash
x=10           # x0

function f() {
    x=20       # x is dynamically bound to x1
}

function g() {
    local x=30 # x1
    f
}

g
echo $x # prints 10
```

Dynamic Scope to Static Scope

```lisp
(let ((x 10))
  (defun f ()
    (setq x 20))
  (defun g ()
    (let ((x 30))
      (f)))
  (g)
  x
)
```

- Dynamic Scope in Emacs Lisp: `x=10` at the end
- Static Scope in Common Lisp: `x=20` at the end

Static vs. Dynamic Scope: Perl

- Dynamic Scope in Perl: `local`

```perl
local $x = 10;
sub f {
    $x = 20;
}
sub g {
    local $x = 30;
    f ();
}
g ();
print ($x); # prints 10
```

- Static Scope in Perl: `my`

```perl
my $x = 10;
sub f {
    $x = 20;
}
sub g {
    my $x = 30;
    f ();
}
g ();
print ($x); # prints 20
```

Shadowing in Statically Scoped Languages

- Static scope at the level of blocks

  ❓ Should reusing names be allowed?

```
static void f () {
  int x = 1;
  {
    int y = x + 1;
    {
      int x = y + 1;
      System.out.println ("x = " + x);
    }
  }
}
```

- In Java, shadowing of local variables is not allowed, see [Java Language Specification]

```
$ javac C.java
C.java:7: error: variable x is already defined in method f()
        int x = y + 1;
            ^
1 error
```

Shadowing

- In Java, local variables are allowed to shadow fields

```java
public class C {
  static int x = 1;

  static void f () {
    int y = x + 1;
    {
      int x = y + 1;
      System.out.println ("x = " + x);
    }
  }

  public static void main (String[] args) {
    f ();
  }
}
```

```
$ javac C.java
$ java C
x = 3
```

Shadowing

- C allows shadowing of variables

```c
int main () {
  int x = 1;
  {
    int y = x + 1;
    {
      int x = y + 1;
      printf ("x = %d\n", x);
    }
  }
}
```

```
$ gcc -o scope scope.c
$ ./scope
x = 3
```

- Scala allows shadowing of variables

```scala
def main (args:Array[String]) =
  var x = 1
    var y = x + 1
      var x = y + 1
      println ("x = " + x)
end main
```

- Indentation creates blocks!

```
$ scalac C.scala
$ scala C
x = 3
```

Shadowing and Recursion

Is `x` in scope?

- C warns about uninitialized variables

```c
1 int main (void) {
2   int x = 10;
3   {
4     int x = x + 1;
5     printf ("x = %08x\n", x);
6   }
7   return 0;
8 }
```

```
$ gcc -o scope scope.c

$ gcc -Wall -o scope scope.c
scope.c: In function ‘main’:
scope.c:5:7: warning: unused variable ‘x’ [-Wunused-variable]
scope.c:7:9: warning: ‘x’ is used uninitialized in this function [-Wuninitialized]

$ ./scope
x = 00000001
```

- Java requires initialized variables

```java
1 public static void main (String[] args) {
2   int x = x + 1;
3   System.out.printf ("x = %08x\n", x);
4 }
```

```
x.java:3: error: variable x might not have been initialized
        int x = x + 1;
                ^
```

Shadowing and Recursion

- Scala variables and fields are set to default values (e.g., `0`) before the initialization code is run
- Recursion is allowed when initializing fields

Scala

```
scala> val x:Int = 1 + x
x: Int = 1
```

Expressed in Java

```java
public class C {
  private final int x; // default-initialized to 0
  public int x() { return x; }
  public C() { x = 1 + x; }
}
```

Shadowing and Recursion

Does that work with complex datatypes?

```scala
val xs:List[Int] = 1 :: xs
// java.lang.NullPointerException
```

- `xs` default-initialized to `null`
- `null != Nil`: exception occurs because `1 :: null` is `null.::(1)`

Scala Lazy Lists

- Lazy lists `#::` are non-strict

```scala
val ones:LazyList[Int] = 1 #:: ones
// ones: LazyList[Int] = LazyList(<not computed>)
```

```
scala> ones.take (5)
res0: scala.collection.immutable.LazyList[Int] = LazyList(<not computed>)

scala> ones.take (5).toList
res1: List[Int] = List(1, 1, 1, 1, 1)
```

Scala Lazy Lists

- *Lazy* evaluation of list elements

```scala
def f (x:Int) : LazyList[Int] =
  println (s"Called f($x)")
  x #:: f(x+1)
end f
// f: (x: Int): LazyList[Int]
```

- Create a lazy list

```
scala> val xs:LazyList[Int] = f(10)
Called f(10)
xs: LazyList[Int] = LazyList(<not computed>)
```

- Access elements of lazy list

```
scala> xs.take(4).toList
Called f(11)
Called f(12)
Called f(13)
res12: List[Int] = List(10, 11, 12, 13)
```

- Access same elements and more

```
scala> xs.take(4).toList
res13: List[Int] = List(10, 11, 12, 13)
```

```
scala> xs.take(6).toList
Called f(14)
Called f(15)
res14: List[Int] = List(10, 11, 12, 13, 14, 15)
```

Recursion and Corecursion

Recursion

- Works **analytically**: breaks computation into smaller pieces until it reaches a base case
- Operates on arbitrarily complex data as long as they can be reduced to simple base cases

Corecursion

- Works **synthetically**: builds up data from a base case
- Generates arbitrarily complex data as long as it can be produced from simple base cases
- Often implemented with lazy evaluation

Example: Recursive and Corecursive Factorial

Recursive

```
def fact(n: Int) : BigInt =
  require(n>=0)
  if n == 0 then 1
  else n * fact(n-1)
```

Tail-Recursive

```
def fact(n: Int, f: Int=1) : BigInt =
  require(n>=0)
  if n == 0 then f
  else fact(n-1, n*f)
```

Corecursive

```
def fact(n: Int=0, f: Int=1) : LazyList[BigInt] =
  require(n>=0)
  f #:: fact(n+1, f*(n+1))
```

Scope and Lifetime

Scope

- Of an identifier in source code
- Where it is bound

Lifetime

- Of an area of memory
- Duration during which it is allocated

Storage Options

Global

- Static storage
- Available for lifetime of program

Call Stack

- *In activation records in call stack* (stack-allocated, recall Systems 1)
- Available whilst function active (called but not returned)

Heap

- *In heap* (heap-allocated)
- Available until deallocated (manually or via garbage collection)

Lifetime Issues

- 🐞 Lifetime too short: common with explicit memory de-allocation
  - reads return other value
  - writes overwrite other value
  - resource state incorrect, e.g., file handle closed
  - can cause security problems
- 🐞 Lifetime too long: common with garbage collection
  - uses too much memory (*memory leak*)
  - too late in freeing other resources / finalization
  - can cause vulnerability to denial of service attacks

Dangling Pointers: Stack

What is wrong with this program?

```c
#include <stdio.h>
#include <stdlib.h>

int *f (int x) {
  int y = x;
  return &y;
}

int main (void) {
  int *p = f (1);
  printf ("*p = %d\n", *p);
  return 0;
}
```

- Compile warning

```
$ gcc -o ar ar.c
ar.c: In function ‘f’:
ar.c:6:3: warning: function returns address of local variable
  [enabled by default]

$ ./ar
*p = 1
```

Dangling Pointers

- Static analysis tools can help

```
$ valgrind ./ar
==5505== Memcheck, a memory error detector
==5505== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==5505== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==5505== Command: ./ar
==5505==
==5505== Conditional jump or move depends on uninitialised value(s)
==5505==    at 0x4E7C1A1: vfprintf (vfprintf.c:1596)
==5505==    by 0x4E85298: printf (printf.c:35)
==5505==    by 0x400536: main (in /tmp/ar)
==5505==
==5505== Use of uninitialised value of size 8
==5505==    at 0x4E7A49B: _itoa_word (_itoa.c:195)
==5505==    by 0x4E7C4E7: vfprintf (vfprintf.c:1596)
==5505==    by 0x4E85298: printf (printf.c:35)
==5505==    by 0x400536: main (in /tmp/ar)
==5505==
==5505== Conditional jump or move depends on uninitialised value(s)
==5505==    at 0x4E7A4A5: _itoa_word (_itoa.c:195)
==5505==    by 0x4E7C4E7: vfprintf (vfprintf.c:1596)
==5505==    by 0x4E85298: printf (printf.c:35)
==5505==    by 0x400536: main (in /tmp/ar)
==5505==

```

```
*p = 1
==5505==
==5505== HEAP SUMMARY:
==5505==     in use at exit: 0 bytes in 0 blocks
==5505==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==5505==
==5505== All heap blocks were freed -- no leaks are possible
==5505==
==5505== For counts of detected and suppressed errors, rerun with: -v
==5505== Use --track-origins=yes to see where uninitialised values come from
==5505== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 2 from 2)
```

Dangling Pointers: Heap

What is wrong with this program?

```c
#include <stdio.h>
#include <stdlib.h>

int *f (int x) {
  int *result = (int *) malloc (sizeof (int));
  *result = x;
  return result;
}

int main (void) {
  int *p = f (1);
  printf ("*p = %d\n", *p);
  return 0;
}
```

- Program compiles

```
$ gcc -Wall -o ar ar.c && ./ar
*p = 1
```

- but…

Dangling Pointers: Heap

```
$ valgrind ./ar
==10962== Memcheck, a memory error detector
==10962== Copyright (C) 2002-2011, and GNU GPL, by Julian Seward et al.
==10962== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==10962== Command: ./ar
==10962==
*p = 1
==10962==
==10962== HEAP SUMMARY:
==10962==     in use at exit: 4 bytes in 1 blocks
==10962==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==10962==
==10962== LEAK SUMMARY:
==10962==    definitely lost: 4 bytes in 1 blocks
==10962==    indirectly lost: 0 bytes in 0 blocks
==10962==      possibly lost: 0 bytes in 0 blocks
==10962==    still reachable: 0 bytes in 0 blocks
==10962==         suppressed: 0 bytes in 0 blocks
==10962== Rerun with --leak-check=full to see details of leaked memory
==10962==
==10962== For counts of detected and suppressed errors, rerun with: -v
==10962== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

Dangling Pointers: Heap

What is wrong with this program?

```c
#include <stdio.h>
#include <stdlib.h>

int *f (int x) {
  int *result = (int *) malloc (sizeof (int));
  *result = x;
  return result;
}

int main (void) {
  int *p = f (1);
  free (p);
  printf ("*p = %d\n", *p);
  return 0;
}
```

- Program compiles

```
$ gcc -Wall -o ar ar.c && ./ar
*p = 0
```

- but…

Dangling Pointers: Heap

```
$ valgrind ./ar
==13594== Memcheck, a memory error detector
==13594== Copyright (C) 2002-2011, and GNU GPLd, by Julian Seward et al.
==13594== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==13594== Command: ./ar
==13594==
==13594== Invalid read of size 4
==13594==    at 0x4005D2: main (in /tmp/ar)
==13594==  Address 0x51f0040 is 0 bytes inside a block of size 4 freed
==13594==    at 0x4C2A82E: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==13594==    by 0x4005CD: main (in /tmp/ar)
==13594==
*p = 1
==13594==
==13594== HEAP SUMMARY:
==13594==     in use at exit: 0 bytes in 0 blocks
==13594==   total heap usage: 1 allocs, 1 frees, 4 bytes allocated
==13594==
==13594== All heap blocks were freed -- no leaks are possible
==13594==
==13594== For counts of detected and suppressed errors, rerun with: -v
==13594== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
```

Summary

Scope

- Where an identifier is visible
- Global scope: variables are accessible from anywhere
- Static scope: closest lexical appearance in source code
- Dynamic scope: closest activation record

Lifetime

- How long a memory location is available
- 🐞 Dangling pointers: point to freed memory
- 🐞 Memory leaks: allocated memory that does not get freed