Implementing Control Flow and Scope

reconciling "the call stack" with "the environment" under the hood

Inline Blocks

```c
{ int x = 2;
 int y = 10
 { int z = 2;
   int x = 3;
   x = z + y;
 } print x;
}
```

Control link

Environment Pointer

Function Calls

```c
1 int squm(int n) {
2   int i, sum = 0;
3   for (i = 0; i < n; i++)
4     sum = sum + i * i;
5   return sum;
6 }
7 void main() {
8   int x = squm(15);
9   print x;
10 }
```
Activation Record stores evaluation context
(CS 240: a.k.a. call frame, stack frame, etc.)

- Control link
- Return address
- Return-result addr
- Parameters
- Local variables
- Intermediate results

Environment
Pointer

Arrangement differs per platform. All parts stored somewhere, may mix registers, memory.

Activation Records

fun fact(n) =
  if n <= 1 then 1
  else fact(n-1)*n
val y = fact(2)

fun force(a) = m * a
fun cow(y) =
  let m = y * y in
  force(m)
end
val _ = cow(10)

Dynamic Scope = follow control links

Stack Inspection (Java/JVM)

Permission is dynamically scoped. Depends on:
  * permission of calling method
  * permission of all transitive callers (methods deeper on stack)

void open(String s) {
  SecurityManager.checkRead();
  ...
}
Stack Inspection (Java/JVM)

Permission is dynamically scoped. Depends on:
- permission of calling method
- permission of all transitive callers (methods deeper on stack)

```java
void open(String s) {
    SecurityManager.checkRead();
    ...
}
```

Fails if Applet code is not trusted.

Accessing outer bindings?
```
val m = 5
fun force(a) = m * a
fun cow(y) = let m = y * y in force(m) end
val _ = cow(10)
```

Lexical Scope = ???

# links to follow?

Control ≠ Environment!
Separate link.
```
val m = 5
fun force(a) = m * a
fun cow(y) = let m = y * y in force(m) end
cow(10)
```

Control link
```
control link
m 5
control link
force ...
control link
cow ...
control link
moo ...
control link
env link
m 5
control link
env link
force ...
control link
cow ...
control link
env link
m 100
control link
env link
cow(10)
control link
force(100)
control link
eenv link
a 100
```
Activation Record for Lexical Scope

- Control link: to activation record of caller
- Environment link: to activation record of closest lexically enclosing scope in program
  - a.k.a. access link, scope link

Difference
- Control structure depends on dynamic behavior of program.
- Environment structure depends on static (lexical) form of program text.

Implementation So Far
- Activation records track separate:
  - Control link: what code called this code/should continue executing next?
  - Environment link: what environment does this activation record extend?
- Closures:
  - Environment reference: to activation record where defined.
  - Code reference: to code
- On function call, new activation record with:
  - Control link set to caller’s activation record.
  - Environment link set to closure’s environment.
- SO FAR: all control/environment links point “back” (deeper) in the stack
  - Can still deallocate activation records in LIFO order
- But what about returning functions...?

Returning a closure...

(* make a unit converter *)
fun make scale = fn x => x * scale
val toIN = make (1.0 / 2.54)
val inches = (toIN 6.0) + (toIN 20.0)
Returning a closure: **broken**

(* make a unit converter *)
fun make scale = fn x => x * scale
val toIN = make (1.0 / 2.54)
val inches = (toIN 6.0) + (toIN 20.0)

LIFO stack of activation records will not work!

One solution: **the "heap-allocated stack"**

(* make a unit converter *)
fun make scale = fn x => x * scale
val toIN = make (1.0 / 2.54)
val inches = (toIN 6.0) + (toIN 20.0)

Give up on stack. Heap-allocate + GC activation records.

Contributes to generational hypothesis.

Free variables: when scope matters

(* xs is a long list *)
fun make x xs =

Recursive definition:
\[ FV(x) = \{x\} \]
\[ FV(e1 + e2) = FV(e1) \cup FV(e2) \]
\[ FV(fn \Rightarrow e) = FV(e) - \{x\} \]

\[ FV(expr) = \text{variables used where not bound within } expr. \]
Inefficiencies of Basic Heap-Allocated Stack

(* xs is a long list *)

fun make x xs =
  let val temp1 = map (fn y => ...) xs
      val temp2 = filter (fn y => ...) temp1
      val (a::rest) = map (fn y => ...) temp1
    in
      fn z => x + a + z
    end
  val f = make 31 [...] 
  val n = f 57

Lots of garbage reachable from closure.

Alternative: save only free-variable bindings

(* xs is a long list *)

fun make x xs =
  let val temp1 = map (fn y => ...) xs
      val temp2 = filter (fn y => ...) temp1
      val (a::rest) = map (fn y => ...) temp1
    in
      fn z => x + a + z
    end
  val f = make 31 [...] 
  val n = f 57

Summary: Implementing Control and Scope

- **Activation records track:**
  - **Control link:** what code called this code/should continue executing next?
  - **Environment link:** what environment does this activation record extend?

- **Closures:**
  - Environment reference: to activation record where defined (or copy of free vars)
  - Code reference: to code

- On function call, new activation record with:
  - Control link set to caller’s activation record.
  - Environment link set to closure’s environment.

- **Cannot manage activation records with stack discipline alone, but:**
  - Heap-allocate the stack or at least the copied closure environments.
  - Either way: Generational GC useful!