CS558 Programming Languages

Fall 2015

Lecture 4

STORAGE LIFETIMES

So far our interpreter-based semantics have assumed a very simplified view of the **store**.

- single homoeneous collection of locations;
- number of available locations is unbounded;
- once allocated, locations are never freed or re-used.

In real language implementations things are quite different:

- machines can (efficiently) provide only a limited amount of memory;
- computations often need to allocate more than that much memory, but not all at once;
- thus, nearly all language implementations support freeing and re-use of locations that are no longer needed.

The **lifetime** of an allocated chunk of memory (loosely, an "object") extends from the moment of allocation to the moment of freeing.

Most higher-level languages provide several different classes of storage locations, classified by their lifetimes.

Storage Classes Based on Data Lifetime

Static Data : Permanent Lifetimes

- Global variables and constants.
- Efficient access (via constant addresses); no allocation/deallocation
- Original FORTRAN (no recursion) used even for local variables.

Stack Data : Nested Lifetimes

- Allocation/deallocation and access are efficient (via stack pointer).
- Good locality for VM systems, caches.
- Most languages use stack to store local variables (and internal control data for function calls).

Heap Data : Arbitrary Lifetimes

- Requires runtime system support for allocation and deallocation or garbage collection.
- Commonly used for storing linked data structures (e.g. objects).
- Languages with first-class functions use heap for free variable values

Scope, Lifetime, and Memory Bugs

Lifetime and scope are closely connected. To avoid memory bugs, each store object must remain live as long as an in-scope identifier might point to it (directly or indirectly).

For stack data, the language implementation normally enforces this automatically.

- A function's local variables are typically bound to locations in a **stack frame** whose lifetime lasts from the time the function is called to the time it returns — exactly when its variables go out of scope for the last time.

For heap data, the requirement is trickier to enforce, unless the language uses a **garbage collector**.

- Most collectors work by recursively **tracing** all objects that are accessible from identifiers currently in scope (or that might come back into scope later during execution).
- Only unreachable objects are deallocated for possible future re-use.

Problems with Explicit Control of Lifetimes

Many older languages support pointers and explicit deallocation of storage, which can be somewhat more efficient than garbage collection.

But explicit deallocation makes it easy for the programmer to accidentally kill off an object even though it is still accessible, e.g. in C++:

```
char *foo() {
  char *p = new char[100]; // allocate
  delete p;                // free
  return p;}               // but remember
```

Here the allocated storage remains accessible (via the value of variable `p`) even after that storage has been freed (and possibly reallocated for something else).

This is a **dangling pointer** bug. These can be very hard to find, because their effects are typically not visible until (arbitrarily) later in execution.

A **space leak** bug results from the converse mistake, failing to deallocate an object that is no longer needed. It can make a a program fail unnecessarily by running out of memory.

PROCEDURE ACTIVATION DATA

Each invocation of a procedure requires associated data, such as:

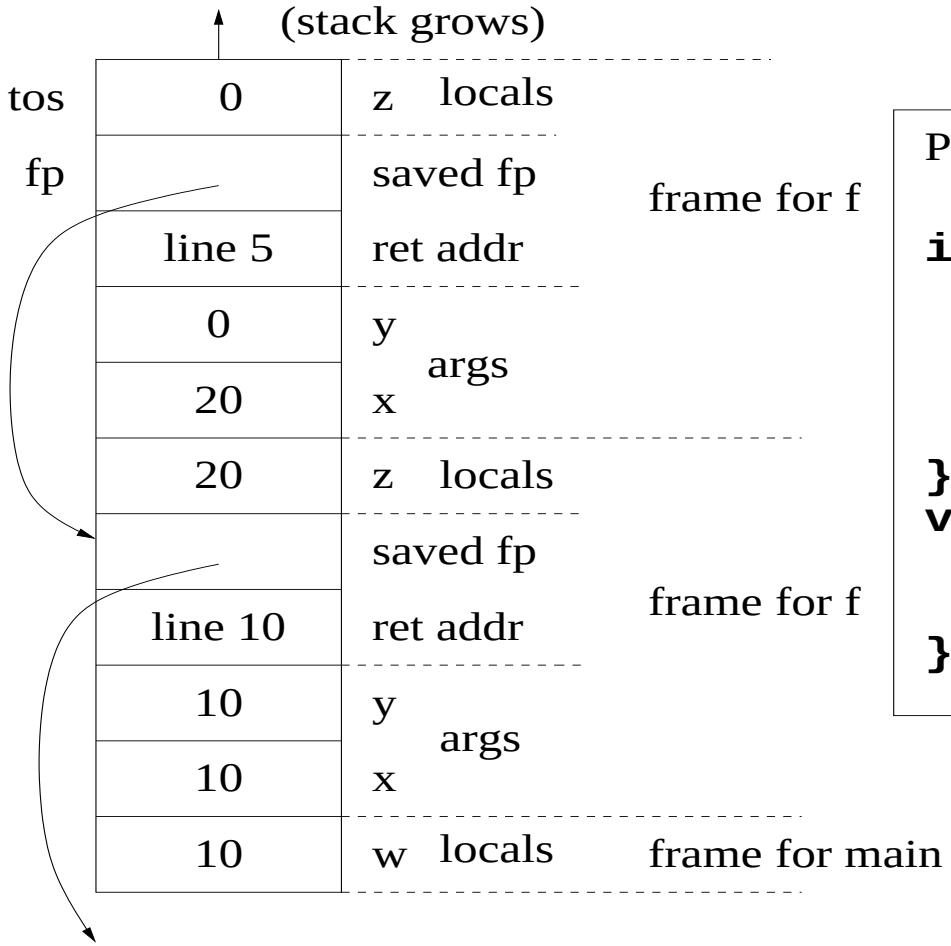
- the **actual** values corresponding to the **formal** parameters of the procedure
- space for the values of **local variables** associated with the procedure.
- the **return address** of the caller

This activation data must live from the time the procedure is invoked until the time it returns. If one procedure calls another procedure, their activation data must be kept separate, because their lifetimes overlap. In particular, each invocation of a recursive procedure needs a separate set of activation data.

ACTIVATION STACKS

For most languages, activation data can be stored on a **stack**, and we speak of pushing and popping activation **frames** from the stack, which is a very efficient way of managing local data.

A typical activation stack, shown just before inner call to `f` returns.



Program:

```
int f(int x, int y){
    int z = y+y;
    if (z > 0)
      z = f(z,0);
    return z+y;
}
void main() {
    int w = 10;
    w = f(w,w);
}
```

WHAT ABOUT REGISTERS?

Although it is convenient to view all locations as memory addresses, most machines also have **registers**, which are:

- much faster to access,
- but very limited in number (e.g., 4 to 64).

So compilers try to keep variables (and pass parameters) in registers when possible, but always need memory as a backup. Using registers is fundamentally just an (important!) optimization.

Easy to have environment map each name to location that is **either** memory address or register.

- But registers don't have addresses, so they can't be accessed indirectly, and register locations can't be passed around or stored.

ITERATION VS. RECURSION

Any iteration can be written as a recursion.

For example, in Scala:

```
while (c) {e}
```

is equivalent to

```
def f(b:Boolean):Unit =
  if (b) {
    e;
    f(c)
  }
f(c)
```

where we assume that the variables used by c and e are in scope.

When can we do the converse? It turns out that a recursion can be rewritten as an iteration (without needing any extra storage) whenever all the recursive calls are in **tail position**. To be in tail position, the call must be the **last** thing performed by the caller before it itself returns.

SCALA TAIL-CALL EXAMPLES

List operations can often be made tail-recursive in this way:

```
def find (y:Int,xs:List[Int]):Boolean = xs match {
   case Nil => false
   case (x::xs1) => (x == y) || find(y,xs1) // tail-recursive
   }

def length (xs:List[Int]):Int = xs match {
   case Nil => 0
   case (_::xs1) => 1 + length(xs1)  // not tail-recursive
   }

def length_tr (xs:List[Int]):Int = {
  // use an auxiliary function with an accumulating parameter
  def f (xs:List[Int],len:Int):Int = xs match {
          case Nil => len
          case (_::xs1) => f (xs1,len+1) // tail-recursive
          }
  f(xs,0)
}
```

A decent compiler can turn tail-calls into iterations, thus saving the cost of pushing an activation frame on the stack. This is essential for functional languages, and useful even for imperative ones.

SYSTEMATIC REMOVAL OF RECURSION

(Adapted from Sedgewick, *Algorithms*, 2nd ed.. Examples in C.)

But what about general (non-tail) recursion? One way to get a better appreciation for how recursion is implemented is to see what is required to get rid of it.

Original program:

```
typedef struct tree *Tree;
struct tree {
  int value;
  Tree left, right;
};

void printtree(Tree t) {
  if (t) {
    printf("%d\n",t->value);
    printtree(t->left);
    printtree(t->right);
  }
}
```

STEP 1

Remove **tail-recursion**.

```
void printtree(Tree t) {
top:
  if (t) {
    printf("%d\n",t->value);
    printtree(t-left);
    t = t->right;
    goto top;
  }
}
```

STEP 2

Use explicit stack to replace non-tail recursion. Simulate behavior of compiler by pushing local variables and return address onto the stack **before** call and popping them back off the stack **after** call.

Assume this stack interface:

```
Stack empty;
void push(Stack s,void* t);
(void*) pop(Stack s);
int isEmpty(Stack s);
```

STEP 2 (CONT.)

Here there is only one local variable (`t`) and the return address is always the same, so there's no need to save it.

```
void printtree(Tree t) {
  Stack s = empty;
top:
  if (t) {
    printf("%d\n",t->value);
    push(s,t);
    t = t->left;
    goto top;
retaddr:
    t = t->right;
    goto top;
  }
  if (!(isEmpty(s))) {
    t = pop(s);
    goto retaddr;
  }
}
```

STEP 3

Simplify by:

- Rearranging to avoid the `retaddr` label.
- Pushing right child instead of parent on stack.
- Replacing first `goto` with a `while` loop.

```
void printtree(Tree t) {
  Stack s = empty;
top:
  while (t) {
    printf("%d\n",t->value);
    push(s,t->right);
    t = t->left;
  }
  if (!(isEmpty(s))) {
    t = pop(s);
    goto top;
  }
}
```

STEP 4

Rearrange some more to replace outer `goto` with another `while` loop.

(This is slightly wasteful, since an extra `push`, `stackempty` check and `pop` are performed on root node.)

```
void printtree(Tree t) {
  Stack s = empty;
  push(s,t);
  while(!(isEmpty(s))) {
    t = pop(s);
    while (t) {
      printf("%d\n",t->value);
      push(s,t->right);
      t = t->left;
    }
  }
}
```

STEP 5

A more symmetric version can be obtained by pushing and popping the left children too.

Compare this to the original recursive program.

```
void printtree(Tree t) {
  Stack s = empty;
  push(s,t);
  while(!(isEmpty(s))) {
    t = pop(s);
    if (t) {
      printf("%d\n",t->value);
      push(s,t->right);
      push(s,t->left);
    }
  }
}
```

STEP 6

We can also test for empty subtrees **before** we push them on the stack rather than after popping them.

```
void printtree(Tree t) {
  Stack s = empty;
  if (t) {
    push(s,t);
    while(!(isEmpty(s))) {
      t = pop(s);
      printf("%d\n",t->value);
      if (t->right)
        push(s,t->right);
      if (t->left)
        push(s,t->left);
    }
  }
}
```

MODELING STACK STORAGE

We can modify our Variables interpreter to treat the store as a data stack.

- Only parameters and local variables are allocated storage locations
- When we leave the scope of a parameter or local variable we deallocate its location.
- Because of the lexical nesting of scopes, this is always the most recently allocated location still in the store.

The location can then be reused on the next allocation (just like in a hardware stack).

```
case class Store(nextAddr:Location, contents:Map[Location,Value]) {
  def allocate(n:Int) : (Location,Store) =
        (nextAddr,Store(nextAddr+n,contents))
  def pop(n:Int):Store =
        Store(nextAddr-n, contents--List.range(nextAddr-n,nextAddr))
  ...
}
```

```
def interpE(expr:Expr,env:Env,st:Store) : (Value,Store) = expr match {
  ...
  case App(f,es) => interpE(f,env,st) match {
    case (ClosureV(xs,b,cenv),st1) =>
      ...
      val (a,st3) = st2 allocate (xs.length)
      val as = List.range(a,a+xs.length)
      val st4 = (st3 /: (as zip vs)) (_+_)
      val (v,st5) = interpE(b,cenv ++ (xs zip as),st4)
      (v,st5 pop (xs.length))
    ...
  }
  case Let(x,d,b) => {
    val (v,st1) = interpE(d,env,st)
    val (a,st2) = st1 allocate 1
    val st3 = st2 + (a -> v)
    val (vb,st4) = interpE(b,env + (x -> a),st3)
    (vb,st4 pop 1)
  }
}
```

PROBLEMS WITH FIRST-CLASS FUNCTIONS

But this interpreter doesn't work for first-class functions!

Consider this program (which uses a "Curried" function; see lecture 2a).

```
(let f (fun x
           (fun y
              (+ x y)))
  (@ (@ f 1) 2))
```

For simplicity, we assume all functions take exactly one parameter.

Here `f` is a function that takes a parameter `x` and **returns** a closure for the (anonymous) function `(fun y (+ x y))`.

We apply `f` to `1` and then apply the resulting function to `2`.

Tracing execution of the applications after binding `f`, we see a sequence like this...

Action	*Map*	*Store*
Evaluate `(@ (@ f 1) 2)`	$\{f \mapsto 0\}$	1 $\{0 \mapsto C(\texttt{x}, \texttt{(fun y (+ x y))}, \emptyset)\}$
Evaluate `(@ f 1)`	$\{f \mapsto 0\}$	
Evaluate `f`	$\{f \mapsto 0\}$	
Yields $C(\texttt{x}, \texttt{(fun y (+ x y))}, \emptyset)$		
Evaluate `1`	$\{f \mapsto 0\}$	
Yields $N(1)$		
Bind formal parameter `x`	$\{x \mapsto 1\}$	2 $\left\{\begin{array}{l} 0 \mapsto C(\texttt{x}, \texttt{(fun y (+ x y))}, \emptyset) \\ 1 \mapsto N(1) \end{array}\right\}$
Evaluate `(fun y (+ x y))`	$\{x \mapsto 1\}$	
Yields $C(\texttt{y}, \texttt{(+ x y)}, \{x \mapsto 1\})$		
Pop formal parameter `x`	$\{f \mapsto 0\}$	1 $\{0 \mapsto C(\texttt{x}, \texttt{(fun y (+ x y))}, \emptyset)\}$
Yields $C(\texttt{y}, \texttt{(+ x y)}, \{x \mapsto 1\})$		
Evaluate 2	$\{f \mapsto 0\}$	
Yields $N(2)$		
Bind formal parameter `y`	$\{x \mapsto 1, y \mapsto 1\}$	2 $\left\{\begin{array}{l} 0 \mapsto C(\texttt{x}, \texttt{(fun y (+ x y))}, \emptyset) \\ 1 \mapsto N(2) \end{array}\right\}$
Evaluate `(+ x y)`	$\{x \mapsto 1, y \mapsto 1\}$	
Evaluate `x`	$\{x \mapsto 1, y \mapsto 1\}$	
Yields $N(2)$		
Evaluate `y`	$\{x \mapsto 1, y \mapsto 1\}$	
Yields $N(2)$		
Yields $N(4)$		

First-class Functions need the Heap

By the time we apply the returned function, the stack activation record containing its free variables may have been popped.

Bottom Line: If we want the flexibility of fully first-class functions, we cannot store their free variables on the stack; they must live in the heap.

How to arrange this?

- Simple solution: Just allocate **all** activation records in the heap instead of the stack, and rely on garbage collection to deallocate and re-use them. (A few compilers do this.)

- More refined solution: Store just those variables that might be **free** in some closure expression in the heap. This is usually done by adding an extra layer of boxing around such variables. (Some compilers do this.)

- Functional language solution: If the free "variables" are actually immutable, we can store **copies** of their values in a heap closure record. (Most functional language compilers do this.)

THE TYRANNY OF THE STACK

Many older languages support functions as values, but not in a fully first-class way—usually because the language designers wanted to avoid using the heap to store variables.

For example, some languages with nested functions (e.g. Pascal, Modula, Ada) allow functions to be passed as **parameters** to other functions, but not returned or stored.

- Under this restriction, if function `f` defines nested first-class function `g`, it is imposible for `g` to be applied after `f` has returned. (Why?)

- So the compiler to use conventional stack activation records, because every free variable of a function lives in an activation record that is still on the stack when the function might be called.

On the other hand, C/C++ allow fully first-class functions, but these languages do not have nested functions, so functions never have free variables, and closures become just simple **function pointers**.

TAIL-CALLS REVISITED

Consider again the non-tail-recursive `length` function. We wrap it in a function that prints out the result:

```
def foo() = {
  def length (xs:List[Int]):Int = xs match {
     case Nil => 0
     case (_::xs1) => 1 + length(xs)  // not tail-recursive
     }
  println(length(List(19,42)))
}
```

We can write this in a tail-recursive way like this:

```
def foo() = {
  def klength(xs:List[Int],k:Int=>Unit):Unit = xs match {
     case Nil => k(0)
     case (_::xs1) => klength(xs1,i => k(i+1))
     }
  klength(List(19,42),println)
}
```

COMPARISON OF COMPUTATIONS

This rather odd code was constructed by giving `klength` an additional argument, `k`, of type `Int => Unit`. Instead of returning its "result" value, `klength` passes it to `k`.

```
println (length (List(19,42))) →
println (length (List(42)) + 1) →
println ((length (Nil) + 1) + 1) →
println ((0 + 1) + 1) →
println (1 + 1) →
println (2)
```

```
klength (List(19,42), println)→
klength (List(42), i1 => println(i1 + 1)) →
klength (Nil, i2 => (i1 => println(i1 + 1)) (i2 + 1)) →
(i2 => (i1 => println(i1 + 1))(i2 + 1)) (0) →
(i1 => println(i1 + 1)) (0 + 1) →
(i1 => println(i1 + 1)) (1) →
println (1 + 1)  →
println (2)
```

CONTINUATION-PASSING STYLE

Notice that **every** call is now a **tail-call**. Note too that `klength` only returns after `println` is invoked. If it were the whole program, it wouldn't need to return at all. This is because `k` is serving the same role as a return address: saying "what to do next."

This means we can evaluate `klength` **without a stack**!

Functions like `k` are called **continuations** and programs written using them are said to be in **continuation-passing style (CPS)**.

We may choose to write (parts or all of) programs explicitly in CPS because it makes it easy to express a particular algorithm or because it clarifies the control structure of the program.

Note that CPS programs are just a subset of ordinary functional programs that happens to make heavy use of the (existing) enormous power of first-class functions. Remarkably, we can also systematically convert **any** functional program into an equivalent CPS program. (Details omitted.)

CONTINUATIONS

More broadly speaking, the term **continuation** means any representation of the program's state at a particular point during execution. This state must contain exactly the information needed to "continue" execution of the program from that point until the program completes.

CPS programs represent each continuation as a **function**, which, given the **value** of the expression being computed at that point, returns the result of the entire program.

If we are describing execution in terms of a low-level machine, we can think of a continuation for a program point as the machine state at that point: the values of the pc, return stack, etc. These contain the same information that would go into a closure for the continuation function.