Lecture #25: Achieving Runtime Effects—Functions

Administrivia
• Proj3 Java files (mostly) out (need some testing stuff).
• Deadline for Project 3 will be pushed a bit due to delays (not too much, because of ACM Programming Contest).

General Considerations
• Language design and runtime design interact. Semantics of functions make good example.
• Levels of function features:
  1. Plain: no recursion, no nesting, fixed-sized data with size known by compiler.
  2. Add recursion.
  3. Add variable-sized unboxed data.
  4. Allow nesting of functions, up-level addressing.
  5. Allow function values w/ properly nested accesses only.
  6. Allow general closures.
  7. Allow continuations.
• Tension between these effects and structure of machines:
  - Machine languages typically only make it easy to access things at addresses like $R + C$, where $R$ is an address in a register and $C$ is a relatively small integer constant.
  - Therefore, fixed offsets good, data-dependent offsets bad.

1: No recursion, no nesting, fixed-sized data
• Total amount of data is bounded, and there is only one instantiation of a function at a time.
• So all variables, return addresses, and return values can go in fixed locations.
• No stack needed at all.
• Characterized FORTRAN programs in the early days.
• In fact, can dispense with call instructions altogether: expand function calls in-line. E.g.,
\[
\text{def } f(x): \\
x *= 42 \\
y = 9 + x; \quad \Rightarrow \text{becomes} \Rightarrow \\
g(x, y) \\
f(3) \\
\]
• However, program may get bigger than you want. Typically, one in-lines only small, frequently executed functions.

2: Add recursion
• Now, total amount of data is unbounded, and several instantiations of a function can be active simultaneously.
• Calls for some kind of expandable data structure: a stack.
• However, variable sizes still fixed, so size of each activation record (stack frame) is fixed.
• All local-variable addresses and the value of dynamic link are known offsets from stack pointer, which is typically in a register.

```
// Top of stack
f's locals
ra
arguments to f

// Base of 1st frame
g's locals
ra
arguments to g

// Stack
f's locals
ra

; 
```

Last modified: Mon Nov 27 15:02:10 2006
### Calling Sequence when Frame Size is Fixed

- So dynamic links not really needed.
- Suppose \( f \) calls \( g \) calls \( f \), as at right.
- When called, the initial code of \( g \) (its prologue) decrements the stack pointer by the size of \( g \)'s activation record.
- \( g \)'s exit code (its epilogue):
  - increments the stack pointer by this same size,
  - pops off the return address, and
  - branches to address just popped. to it.

### 3: Add Variable-Sized Unboxed Data

- "Unboxed" means "not on heap."
- Boxing allows all quantities on stack to have fixed size.
- So Java implementations have fixed-size stack frames.
- But does cost heap allocation, so some languages also provide for placing variable-sized data directly on stack ("heap allocation on the stack")
- \texttt{alloca} in C, e.g.
- Now we do need dynamic link (DL).
- But can still insure fixed offsets of data from frame base (frame pointer) using pointers.
- To right, \( f \) calls \( g \), which has variable-sized unboxed array (see right).

### Other Uses of the Dynamic Link

- Often use dynamic link even when size of AR is fixed.
- Allows use of same strategy for all ARs, simplifies code generation.
- Makes it easier to write general functions that unwind the stack (i.e., pop ARs off, thus returning).

### 4: Allow Nesting of Functions, Up-Level Addressing

- When functions can be nested, there are three classes of variable:
  a. Local to function.
  b. Local to enclosing function.
  c. Global
- Accessing (a) or (c) is easy. It's (b) that's interesting.
- Consider (in Pyth or Python):
  ```python
def f ():
  y = 42  # Local to f
def g (a, q):
  if n == 0: return q+y
  else: return g (n-1, q*2)
```
- Here, \( y \) can be any distance away from top of stack.
Static Links

- To overcome this problem, go back to environment diagrams!
- Each diagram had a pointer to lexically enclosing environment
- In Pyth example from last slide, each 'g' frame contains a pointer to the 'f' frame where that 'g' was defined: the static link (SL)
- To access local variable, use frame-base pointer (or maybe stack pointer).
- To access global, use absolute address.
- To access local of nesting function, follow static link once per difference in levels of nesting.

5: Allow Function Values, Properly Nested Access

- In C, C++, no function nesting.
- So all non-local variables are global, and have fixed addresses.
- Thus, to represent a variable whose value is a function, need only to store the address of the function's code.
- But when nested functions possible, function value must contain more.
- When function is finally called, must be told what its static link is.
- Assume first that access is properly nested: variables accessed only during lifetime of their frame.
- So can represent function with address of code + the address of the frame that contains that function's definition.
- It's environment diagrams again!!
6: General Closures

- What happens when the frame that a function value points to goes away?
- If we used the previous representation (#5), we'd get a **dangerous pointer** in this case:

```python
def incr(n):
    delta = n
def f(x):
    return delta + x
return f
p2 = incr(2)
print p2(3)
```

![Value of incr(2)](image)

```
Value of incr(2)
```

- `incr's` frame
  - `with`
    - `delta`
  - `code for f`
  - `Value of incr(2)`

```
Delta extraneous from incr(2)
```

- `code for f`

```
delta, & n
```

7: Continuations

- Suppose function return were not the end?

```python
def f(cont): return cont
x = 1
def g(n):
    global x, c
    if n == 0:
        print "a", x, n,
        c = call_with_continuation(f)
        print "b", x, n,
    else:
        g(n-1);
        print "c", x, n,
        g(2); x += 1;
        print; c()
# Prints:
# a 1 0 b 10 c 11 c 12
# b 20 c 21 c 22
# b 30 c 31 c 32
...```

- The **continuation**, `c`, passed to `f` is "the function that does whatever is supposed to happen after I return from `f`."
- Can be used to implement exceptions, threads, co-routines.
- Implementation? Nothing much for it but to put all activation frames on the heap.
- **Distributed cost.**
- However, we can do better on special cases like exceptions.

Representing Closures

- Could just forbid this case (as some languages do):
  - Algol 68 would not allow pointer to `f` (last slide) to be returned from `incr`.
  - Or, one could allow it, and do something random when `f` (i.e. via `delta`) is called.
- Scheme and Python allow it and do the right thing.
- But must in general put local variables (and a static link) in a record on the heap, instead of on the stack.
- Now frame can disappear harmlessly.

```
```

Summary

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Plain: no recursion, no nesting, fixed-sized data with size known by compiler, first-class function values.</td>
<td>Use inline expansion or use static variables to hold return addresses, locals, etc.</td>
</tr>
<tr>
<td>2. #1 + recursion</td>
<td>Need stack.</td>
</tr>
<tr>
<td>3. #2 + Add variable-sized unboxed data</td>
<td>Need to keep both stack pointer and frame pointer.</td>
</tr>
<tr>
<td>4. #3 - first-class function values + Nested functions, up-level addressing</td>
<td>Add static link or global display.</td>
</tr>
<tr>
<td>5. #4 + Function values w/ properly nested accesses: functions passed as parameters only.</td>
<td>Static link, function values contain their link. (Global display doesn't work so well)</td>
</tr>
<tr>
<td>6. #5 + General closures: first-class functions returned from functions or stored in variables</td>
<td>Store local variables and static link on heap.</td>
</tr>
<tr>
<td>7. #6 + Continuations</td>
<td>Put everything on the heap.</td>
</tr>
</tbody>
</table>