Lecture #24: Achieving Runtime Effects—Functions

- 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.,

```python
def f(x):
    x *= 42
    y = 9 + x;
    g(x, y)  
    >>> becomes >>>
    x_1 = 3
    x_1 *= 42
    y_1 = 9 + x_1
    g(x_1, y_1)

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.

• So dynamic links not really needed. Suppose $f$ calls $g$ calls $f$, as at right.

• If stack pointer in register, all variables and next frame are at known offsets from stack pointer.
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”)
- alloca in C, e.g.
- Now we need dynamic link (DL).
- Can still insure fixed offsets of data from frame base (frame pointer).
- Suppose f calls g, which has variable-sized unboxed array (see right).
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 recent Python):
  ```python
  def f ():
    y = 42  # Local to f
  def g (n, 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 the 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.
The Global Display

- Historically, first solution to nested function problem used an array indexed by call level, rather than static links.

  ```python
def f0 () :
    q = 42
    def f1 () :
        def f2 () : ... g2 () ...
        def g2 () : ... g1 () ...
        def g1 () : ... f1 () ...
  ```

- Each time we enter a function at lexical level $k$ (i.e., nested inside $k$ functions), save pointer to its frame base in DISPLAY[$k$]; restore on exit.

- Access variable at lexical level $k$ through DISPLAY[$k$].

- Relies heavily on scope rules and proper function-call nesting.
The Global Display

• Historically, first solution to nested function problem used an array indexed by call level, rather than static links.

```python
def f0():
    q = 42
def f1():
    def f2(): ... g2() ...
    def g2(): ... g1() ...
    def g1(): ... f1() ...
```

• Each time we enter a function at lexical level $k$ (i.e., nested inside $k$ functions), save pointer to its frame base in $\text{DISPLAY}[k]$; restore on exit.

• Access variable at lexical level $k$ through $\text{DISPLAY}[k]$.

• Relies heavily on scope rules and proper function-call nesting.
The Global Display

- Historically, first solution to nested function problem used an array indexed by call level, rather than static links.

```python
def f0 ():
    q = 42
    def f1 ():
        def f2 (): ... g2 () ...
        def g2 (): ... g1 () ...
        def g1 (): ... f1 () ...
```

- Each time we enter a function at lexical level $k$ (i.e., nested inside $k$ functions), save pointer to its frame base in DISPLAY[$k$]; restore on exit.

- Access variable at lexical level $k$ through DISPLAY[$k$].

- Relies heavily on scope rules and proper function-call nesting

![Diagram showing nested function frames and DISPLAY array]
The Global Display

• Historically, first solution to nested function problem used an array indexed by call level, rather than static links.

    def f0 ():
        q = 42
    def f1 ():
        def f2 (): ... g2 () ...
        def g2 (): ... g1 () ...
        def g1 (): ... f1 () ...

• Each time we enter a function at lexical level \( k \) (i.e., nested inside \( k \) functions), save pointer to its frame base in DISPLAY[\( k \)]; restore on exit.

• Access variable at lexical level \( k \) through DISPLAY[\( k \)].

• Relies heavily on scope rules and proper function-call nesting
The Global Display

- Historically, first solution to nested function problem used an array indexed by call level, rather than static links.

```python
def f0():
    q = 42
def f1():
    def f2(): ... g2() ...
    def g2(): ... g1() ...
    def g1(): ... f1() ...
```

- Each time we enter a function at lexical level $k$ (i.e., nested inside $k$ functions), save pointer to its frame base in DISPLAY[$k$]; restore on exit.

- Access variable at lexical level $k$ through DISPLAY[$k$].

- Relies heavily on scope rules and proper function-call nesting.
The Global Display

- Historically, first solution to nested function problem used an array indexed by call level, rather than static links.

```python
def f0():
    q = 42
    def f1():
        def f2(): ... g2() ...
        def g2(): ... g1() ...
        def g1(): ... f1() ...
```

- Each time we enter a function at lexical level $k$ (i.e., nested inside $k$ functions), save pointer to its frame base in DISPLAY[$k$]; restore on exit.

- Access variable at lexical level $k$ through DISPLAY[$k$].

- Relies heavily on scope rules and proper function-call nesting.
The Global Display

• Historically, first solution to nested function problem used an array indexed by call level, rather than static links.

```python
def f0 ():
    q = 42
def f1 ():
    def f2 (): ... g2 () ... 
def g2 (): ... g1 () ... 
def g1 (): ... f1 () ... 
```

• Each time we enter a function at lexical level $k$ (i.e., nested inside $k$ functions), save pointer to its frame base in DISPLAY[$k$]; restore on exit.

• Access variable at lexical level $k$ through DISPLAY[$k$].

• Relies heavily on scope rules and proper function-call nesting.