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 \textit{good}, data-dependent offsets \textit{bad}.

2: Add recursion
Now, total amount of data is un-bounded, 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.

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.,

\[
\begin{align*}
def f(x): & \quad x_1 = 3 \\
& \quad x_1 *= 42 \\
& \quad y_1 = 9 + x_1 \\
& \quad g(x, y) \quad \Rightarrow \quad \text{becomes} \quad y_1 = 9 + x_1 \\
& \quad g(x_1, y_1) \\
& \quad f(3)
\end{align*}
\]

However, program may get bigger than you want. Typically, one in-lines only small, frequently executed functions.

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")

\[
\begin{align*}
\text{alloca} \text{ in } C, \text{ e.g.} \\
\text{Now we need dynamic link (DL).} \\
\text{Can still insure fixed offsets of data from frame base (frame pointer).} \\
\text{Suppose } f \text{ calls } g, \text{ which has variable-sized unboxed array (see right).}
\end{align*}
\]
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.

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