We continue looking at implementation of higher-order functions with lexical scoping and other functional programming language features. As described in the previous lecture, we represent function values as closures that bind together the machine code with an environment containing free variables accessed from outer scopes. An example will help show how this works.

1 Escaping variables

We start with a function that applies other functions twice:

```
twice(f: int->int): int->int {
    g(x: int): int {
        return f(f(x))
    }
    return g
}
```

Using this function, we write a function `double` to double numbers:

```
// returns 2*n the hard way
double(n: int): int {
    addn(x: int): int {
        return x+n
    }
    plus2: int->int = twice(addn)
    return plus2(0)
}
```

Now consider what happens if we call `double(5)`. The function `twice` is applied to `addn` to construct a new function `plus2` that adds `n` twice to its argument. This function is applied to zero to obtain 2*n. The call tree of the evaluation is as shown:

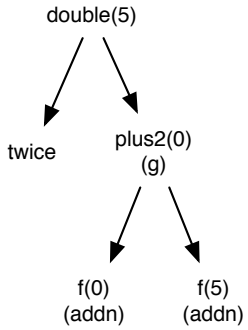

Figure 1 shows what the stack looks like during the call to `twice`. We show all variables on the stack, although some of the variables could be stored in registers. The variables `addn`, `plus2`, and `f` all take up two words because they are closures. The variable `plus2` is uninitialized in the diagram because the assignment to the variable has not yet happened. It is also possible to place closures on the heap and represent a function as a pointer to that memory location; this makes passing closures as arguments or return values cheaper, but creates garbage and takes up more space overall.

The variable `f` belongs to the activation record (i.e., the set of local variables) of `twice`, but it is an escaping variable because it is used by the function `g`, which escapes `twice`. Therefore this part of the activation record is stored on the heap in an *escaping variables record*.

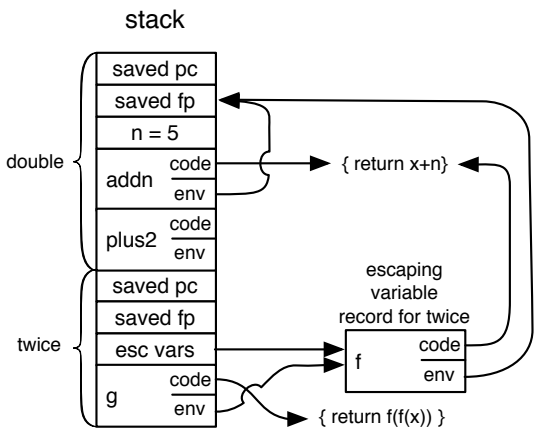

Figure 1: After calling "twice"

By contrast, the variable `n`, needed by `addn`, is not an escaping variable because `addn` does not escape from its enclosing function, `double`. Therefore `n` can be stack allocated, and the environment pointer from the `addn` closure points directly to the stack rather than to the heap.

Figure 2 shows what the stack looks like during the calls to `addn`. When a closure is invoked, the environment pointer is passed as an extra argument, as we saw earlier. This environment pointer becomes a local variable of the function, where it is called the *static link*, as shown in the diagram.

2 Escape analysis

Variables used by escaping functions outlive their stack frame and must therefore be heap-allocated. Since this is more expensive than stack-allocating them, it is useful to perform *escape analysis* to figure out which variables escape and which do not. In general, the activation record of a function consists of a set of escaping variables stored in an escaping variable record, and another set of variables stored on the stack or in registers. Even if some variables escape, we don't want to stack-allocate other variables because that makes them more expensive. Escaping variable records can also pin down objects so that a garbage collector can't collect them, hurting performance.

There is more than one way for a function to escape:

1. It might be returned directly from its enclosing function.
2. It might be stored into a data structure that itself escapes the he encslosing function.
3. It might be passed to another function, either directly as an argument, or indirectly via a data structure, that then stores it into a data structure that outlives the first function.

Doing a good job on escaping variables analysis requires a whole-program analysis. A good pointer analysis does most of the work already. A variable x escapes if it points to something that is known to escape.

3 Static link chains

When functions are nested, static links can form a chain connecting multiple escaping variables records. Consider the functions `f`, `g`, `h`, and `j`, depicted in Figure 3. The functions are nested as shown by the boxes on the left. Function `f` declares variables `x` and `y`, and `h` uses `x` and `j` uses `y`. Suppose that the functions `h` and `j` escape from `f`. The diagram on the right shows how the closures and escaping variables records look.

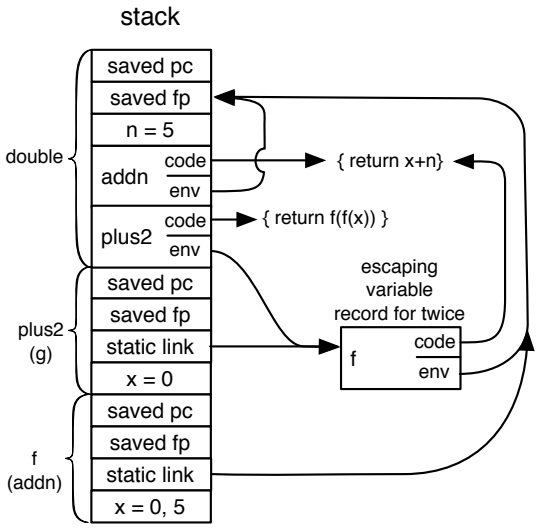

Figure 2: In the calls to "addn"

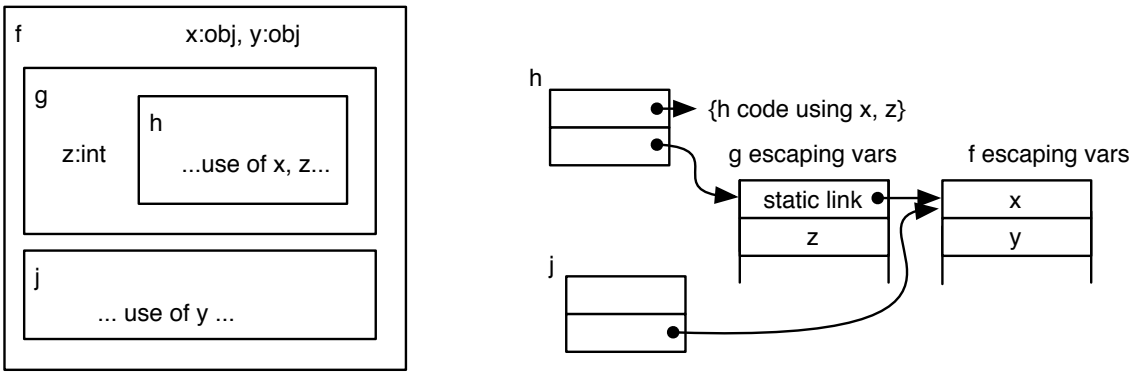

Figure 3: Nested functions and static link chains

Since `h` refers to the variable `x`, it is an escaping variable found in the escaping variables record of `f`. It must be accessed via the static link of `g`, which itself becomes an escaping variable that is stored in `g`'s escaping variable record. In general, when a function like `h` is nested n levels deep, variables are accessed by following a chain of static links. To speed up access to variables in outer scopes, it is possible to copy the pointers to all the outer scopes' activation records into a array within the stack frame. This is called a *display*.[1]

Static link chains create another more subtle problem, however—they can pin down a lot of garbage by making it reachable. For example, consider the escaping variable record for `f`. It cannot be garbage collected as long as *either* the closure for `h` or for `j` are reachable. So even if the `h` closure is garbage-collected, the escaping variable record for `f` will keep objects referenced by `x` from being collected, even though the `j` closure does not use `x`.

One technique that helps with the garbage collection problem is to copy variables from the escaping variable records for outer scopes to the local escaping variable record. This can only be done, however, if the variable is immutable; that is, its value will not change. Precisely identifying such immutable variables requires program analysis in general, though the task is much simplified in languages like ML where all variables are immutable!

4 Tail calls and tail recursion

When programming with higher-order functions, it is important to have good support for *tail recursion*. A function is *tail recursive* if its result is computed by a call to itself. Writing functions in a tail-recursive style is a standard idiom because tail-recursive functions are faster and work better. For example, consider the following two ways of computing the sum of two numbers. (Both are contrived, but are analogous to different ways of writing other recursive functions that traverse lists and other data structures.)

```
// Returns: x+y
// Requires: x>=0
add(x:int, y:int):int {
    if (x == 0) { return y }
    return add(x-1, y+1)
}
// Returns: x+y
// Requires: x>=0
add(x:int, y:int): int {
    if (x == 0) { return y; }
    return 1 + add(x-1, y)
}
```

If you try out these two implementations in your favorite functional language (suitably translated, of course), you'll find that the first version runs much faster than the second one. The second one may cause your computer to run out of memory when `x` is large, whereas the first one will use only a constant amount of memory! The reason is that functional language compilers will translate the first function into, essentially, a `while` loop like this one:

```
while (true) {
    if (x == 0) { return y }
    x = x-1;
    y = y+1;
    continue;
}
```

The `while` loop uses only a constant amount of stack to store its variables, whereas the second implementation of `add` creates $x + 1$ stack frames. The first implementation of `add` behaves just like the `while` loop.

[1] The x86 `enter` instruction is designed to support displays, though this feature is probably not worth trying to use.

Tail recursion is a special case of a *tail call*: a call whose value is immediately returned as the result of the calling function. If we think about what is happening at the assembly language level, a tail call results in code that looks like `call` f`; ret`. The observation is that we replace these two instructions with `jump` f. Since calling functions requires a little more stack maintenance, the actual optimization of a tail call is as follows:

1. Move the arguments to the tail call into the argument registers
2. Restore any callee-save registers
3. Pop the current stack frame
4. Jump directly to the code.

In the case of a tail-recursive tail call, the second and third steps can be avoided because the new stack frame has exactly the same layout as the old one.

5 Lazy evaluation and strictness analysis

Most languages support lazy evaluation in some contexts, but lazy languages are those that avoid computing values until they are needed. Haskell is the most popular lazy language at present.

The opposite of lazy is eager. If we evaluate an expression `f(g(x))` in an eager language like Java or ML, the expression `g(x)` is evaluated to a value before `f` is invoked. In a lazy language, the function `f` is started immediately without evaluating `g(x)`. Thus, lazy language implement *call-by-name* parameter passing in which what is passed is a description of a computation rather than the result of the computation.

For example, if `f` is implemented in the following way:

```
f(y: int):int {
  return y
}
```

then evaluating `f(g(x))` will cause `f` to return a *still*-unevaluated computation.

The actual implementation, called *call-by-need* is more efficient than call-by-name might suggest: instead of passing a computation, a *thunk* containing the computation is passed instead. A thunk allows the computation to be delayed until it is *forced* by evaluating it when it is needed. Once the thunk is forced, the computed value is stored in the thunk, memoizing the computation's result. Subsequent accesses to the thunk will read the memoized value, avoiding recomputation.

Thunks are heap-allocated, so it is good to avoid creating them when they are not needed. A *strictness analysis* determines which arguments to a function are definitely going to be evaluated. The function is said to be *strict* in those arguments. There is no reason not to evaluate the argument expressions before passing them to the function, which substantially speeds up lazy languages.