Lecturer: Andrew Myers

1 Higher-order and first-class functions

Functions (and other kinds of values) are *first-class* if:

1. they can be assigned to variables,
2. they can be passed as function arguments,
3. they can be returned as function results,
4. and they can be stored into data structures

Functions are first-class in quite a few programming languages, especially those considered "functional": ML, Haskell, Scheme, Common Lisp. Many OO languages like JavaScript and Python have first-class functions, and this feature has even been adopted by mainstream languages that did not originally support first-class functions, such as Java and C++.

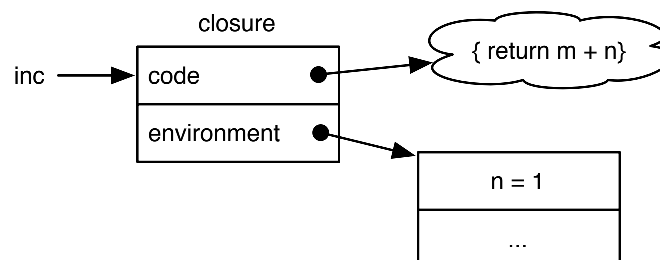
If functions can be passed as arguments or returned as results, we can implement *higher-order functions*. Imagining that we extend Xi with first-class functions, we can implement a function that returns another function:

```
addfun(n: int): int->int {
  f(x: int): int {
    return x+n
  }
  return f
}
inc: int->int = addfun(1)
x: int = inc(5)
// x == 6
```

The variable `inc` is bound here to a function value that is created inside the call to `addfun`. What's happening at the assembly code or IR level? In a language like C, we can represent a function value simply as a pointer to the machine code implementing the function. In this example, the machine code doesn't suffice because the body of `f` has a *free variable* `n` that is not bound by `f`. It comes from an outer scope. Code with unbound variables is called *open* code. How is the code to know that `n` is supposed to be 1 when it is invoked via the variable `inc`?

2 Closures and closure conversion

The standard implementation of a first-class function is as a *closure* that binds together the code of the function with a record containing definitions for its free variables. For example, the closure for `inc` might be represented in memory as shown in this figure:



The closure is a way of representing open code as *closed code* in which there are no free variables. The process of converting open code to closed code is called *closure conversion*. Closure conversion effectively adds an extra environment argument to every function that can be used in a first-class way. For example, the function `f` has a signature like the following when closed (note that `env` represents the memory address of the environment here):

```
f(f_environment env, int x) {
    return x + env.n
}
```

A call to a first-class function, such as `inc(5)`, is then implemented as follows:

```
inc.code(inc.environment, 5)
```

or, in abstract assembly, something like the following:

```
mov rdi, [inc+8]
mov rsi, 5
call [inc]
```

This calling sequence is clearly more expensive than calling the code of `f` directly, but it is generic: it works for any function of type `int->int`. The caller does not need to know the structure of the environment that is being passed.

Note that it is possible to avoid the extra calling cost in the case where the function to be called is known at compile time, and is already closed. In this case the caller and callee can have their own special calling convention—a specialized version of the callee, if you like.

The environment that is part of the closure contains bindings for free variables of the code. We assume *lexical scoping* in which the variables in scope in a given piece of code are apparent in the lexical representation of the program: to find a binding for a variable, we walk outward through the text of the program to find the closest enclosing binding for the variable sought. The most common alternative is *dynamic scoping* in which free variables are evaluated by walking up the heap until a stack frame defining the variable is found. While some languages still do support dynamic scoping (notably Perl), it's usually considered to be bad idea.

Since bindings in the environment mention variables that are in scope in the body of the function, but are not defined in the function, they are variables from *outer scopes* such as containing functions (e.g, `addfun`). Before we introduced first-class functions, variables would be stored on the stack (or, as an optimization, in registers). However, in general, the variables from outer scopes can't be located on the stack any more.

In the `inc` example, the variable `n` is captured by the function returned. Although `f` doesn't assign to `n`, assignments to `n` should still be possible within `f` even after `addfun` returns. And if `addfun` had constructed multiple closures capturing `n`, the different closures should see each other's assignments. So `n` cannot be stored on the stack.

The variable `n` is an example of an *escaping variable*: a variable that is referenced by a function (`f`) that *escapes* the function in which the variable is defined. Escaping variables need to be stored on the heap.

Some languages—notably Pascal—place restrictions on the use of function values, only allowing them to be passed as arguments, but not returned as results or stored into data structures. With this restriction, variables can never escape, and can therefore be stored on the stack. Such first-class function arguments are known as *downward funargs*. Languages that permit more general use of function values have *upward funargs* that may allow variables to escape, and hence must some variables must be stored on the heap.

3 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:

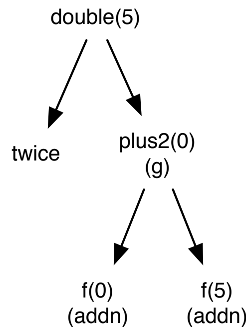


Figure 1 shows what the stack looks like during the call to `twice`. We show all variables on the stack, although some of the variables could be stored in registers. The variables `addn`, `plus2`, and `f` all take up two words because they are closures. The variable `plus2` is uninitialized in the diagram 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*.

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.

4 Static link chains

When functions are nested, static links can form a chain connecting multiple escaping-variable records. Consider the functions `f`, `g`, `h`, and `j`, depicted in Figure 3. The functions are nested as shown by the boxes

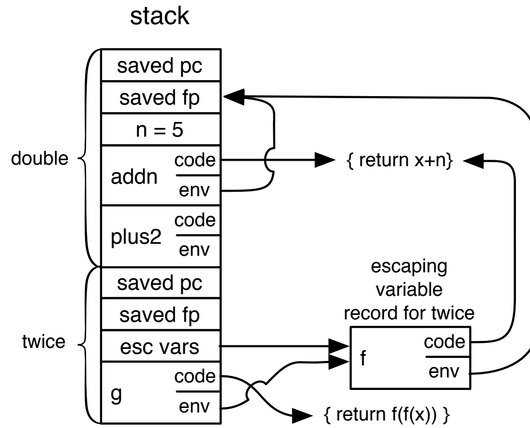


Figure 1: After calling "twice"

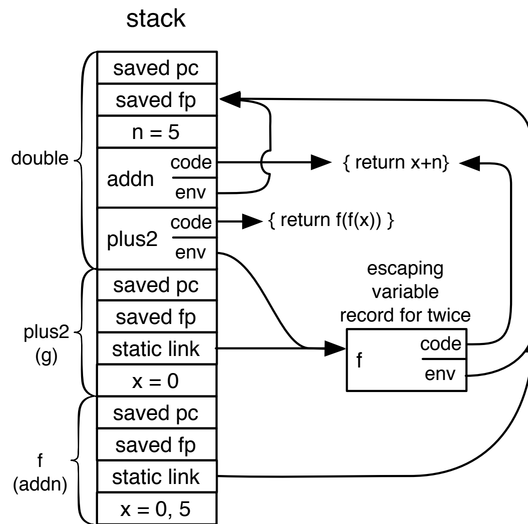


Figure 2: In the calls to "addn"

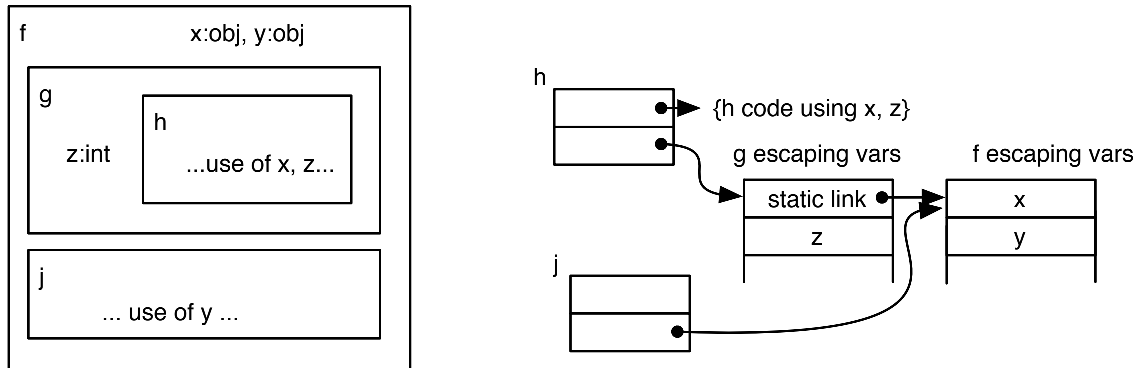


Figure 3: Nested functions and static link chains

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-variable records look.

Since `h` refers to the variable `x`, it is an escaping variable found in the escaping-variable 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!

5 Escape analysis

Variables used by escaping functions outlive their stack frame and must therefore be heap-allocated. Garbage collection and other memory management issues make heap allocation more expensive than stack allocation, so it is useful to perform *escape analysis* to figure out which variables escape and which do not.

In fact, we care about escape analysis for more than just variables. If we can figure out which *objects* do not outlive the function in which they are created, those objects can also be stack-allocated.

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 would still like to stack-allocate other variables (or even allocate them to registers), so they can be accessed more cheaply. Another reason not to have escaping-variable records is that EVRs can pin down objects so that a garbage collector can't collect them, hurting performance.

For escaping variables, we would like to know when a closure of the function defining them escapes. And more generally, we would like to know when other objects can escape.

There is more than one way for a closure or other object to escape:

1. It might be returned directly from its enclosing function.
2. It might be assigned to a global variable.
3. It might be stored into a data structure that itself escapes the enclosing function.
4. It might be passed to another function, either directly as an argument, or indirectly via a data structure, that allows it to escape, perhaps by storing it into a data structure that outlives the first function.

The last item on the list shows that we can't just analyze functions on their own. If the program contains a call like `f(obj)`, we need to know 1) whether `f` allows `obj` to escape and 2) whether `obj` potentially points to something whose escape we care about.

The first requirement shows we need an interprocedural analysis to track the potential escaping objects and variables precisely. The second requirement shows that a pointer analysis (probably also interprocedural) is needed. Therefore, doing a good job at identifying escaping variables requires a whole-program analysis. A good pointer analysis does much of the work already.

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

6 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.

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. The transformation then converts the recursive call into a loop that is subject to further optimization using standard techniques such as loop unrolling and induction variable optimizations.

7 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. In particular, the value of a variable is not computed until that variable is actually used to compute another value. 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 usual implementation of lazy evaluation, 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 is essentially an object that allows the computation to be delayed until it is *forced* by evaluating it when it is needed. Once the thunk is forced, the value it computes 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 and their use involves a conditional that checks at each use whether they have been forced yet. A thunk for a value of type `T` is essentially an object of the following form:

```
struct thunk⟨T⟩ {
  memo: T; // initially null
  code: env → T
  env: environment
}
```

When a thunk `t` of this type is forced to a result, the code sequence is roughly the following:

```
if (t.memo == null)
    t.memo = t.code(t.env)
  result = t.memo
```

Note that the assignment to the memo field needs to be atomic to be thread-safe; in general, locking may be required for multiword results. This code also relies on being able to tell whether the `memo` field has been initialized; if we can't rely on a special value like `null`, we can steal a bit from somewhere else in the structure.

Since thunks are expensive, it is good to avoid creating them when they are not needed, avoiding heap allocation and garbage collection. 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, thus avoiding all the conditional checks at uses and also the creation of the thunk object in the first place. Eagerly evaluating arguments for strict parameters is an important optimization for lazy languages.

8 Variants

Variants, also known as algebraic data types or oneofs, are a core type for functional programming languages. They are a primary way, especially in languages that lack subtyping, of storing more than one kind of type in a variable. In OCaml, we can write a declaration like

`type` t = A_1 `of` t_1 | ... | A_n `of` t_n

Here, the identifiers A_i define *constructors* for the variant type, each expecting a corresponding type t_i. The values of a variant type have the form $A_i(v_i)$ where v_i is a value of type t_i. The types t_i can be omitted, in which case the constructor does not take an argument (technically, that constructor of the variant has *unit* type).

The straightforward implementation of a variant is as a pair of a tag value (representing the constructor A_i) and the value of the corresponding type. Thus, the size of the variant is at least two words when some constructors have an accompanying value. As an optimization for the not uncommon case where all constructors have unit type—where the variant type is merely an enumeration—values of the variant can be just the tag alone.

A particularly important variant type is one representing an `option` or `maybe` type, which offers a safer alternative to programming with `null`. For example, in OCaml we might define such a type as follows:

```
type α maybe = Some of α | None
```

Unfortunately, the simple variant representation described above is not as efficient as programming with a `null` value. Fortunately, the compiler can represent such a type more efficiently. A value `Some(`v`)` is represented as a boxed value containing v, and the value `None` is represented simply as a zero pointer. As long as the language runtime ensures that no boxed value is stored at address zero, the two representations can never be confused.

This implementation is just as efficient as using null pointers. In fact, it can be generalized to allow a larger number of unit-typed constructors:

`type` t `=` A_1 `|` A_2 `| ... |` A_{n-1} `|` A_n `of` t_n

The efficient implementation of this type is to represent a value of constructor A_n as a boxed value of type t_n, and to use distinct illegal addresses for the other constructors $A_1 \ldots A_{n-1}$: for example, distinct small integers, which are normally not used as object addresses in programming-language runtimes. In fact, the optimized representation of enumerations described earlier is just a special case of this representation in which there is no constructor A_n.

For already boxed types t, this implementation is strictly more efficient than the simple one. However, for unboxed types t, the original implementation is probably preferred, because although it takes up more space in the variable or field of maybe type, it avoids an indirection.