EECS 483: Compiler Construction

Lecture 13:
Closures and First-class Functions Continued

February 26
Winter Semester 2025

Announcements

Assignment 3 (Procedures) due on Friday

Spring Break next week

Midterm on Tuesday, March 18, 6-8pm.

- Second week after Spring Break
- Topics: anything covered in assignments 1-3 and lecture material before spring break
- Rooms DOW1010, DOW1017, DOW1018

Midterm review in lecture March 17, bring questions about course material.

After Spring Break:

- optimization
- frontend

Functions as Values

So far in our Snake language, functions are **second class,** meaning that unlike integers/booleans/arrays:

- ordinary program variables cannot be functions

- functions can't be passed as arguments to other functions

- functions can't be returned as values from other functions

This restriction simplifies the job of the compiler, but is uncommon in modern programming languages.

Functions as Values

Modern programming languages allow us to use functions as first-class data

- Low level languages like C/C++ have **function pointers**, which can be passed and returned like any other pointer type

- Higher-level languages both statically (C++, Rust, Java, Go, OCaml, Haskell) and dynamically typed (Python, Ruby, JavaScript, Racket) allow for a more flexible type called **closures**, sometimes called **lambdas**

Used as a convenient interface for implementing iterators, callbacks, concurrency,...

Functions as Values

```
def applyToFive(f):
  f(5)
in

def incr(x):
  x + 1
in

applyToFive(incr)
```

Functions as Values

```
def map(f, a):
  let l = length(a) in
  let a2 = newArray(l) in
  def loop(i):
    if i == l:
      true
    else:
      let _ = a2[i] := f(a[i]) in
      loop(i + 1)
  in
  let _ = loop(0) in
  a2
```

```
def incr(x): x + 1 in
def sqr(x): x * x in
let a = [0, 1, 2] in
map(incr, map(sqr, a))
```

Functions as Values

```
def map(f, a):
  let l = length(a) in
  let a2 = newArray(l) in
  def loop(i):
    if i == l:
      true
    else:
      let _ = a2[i] := f(a[i]) in
      loop(i + 1)
  in
  let _ = loop(0) in
  a2
```

```
let a = [0, 1, 2, 3] in
def sqr_a(i): a[i] := a[i] * a[i] in
let _ map(sqr_a, [1,3]) = in
a
```

need to support variable capture

Lambda Notation

```
def map(f, a):
  let l = length(a) in
  let a2 = newArray(l) in
  def loop(i):
    if i == l:
      true
    else:
      let _ = a2[i] := f(a[i]) in
      loop(i + 1)
  in
  let _ = loop(0) in
  a2
```

```
let a = [0, 1, 2] in
map(lambda x: x + 1 end,
    map(lambda x: x * x end,
    a))
```

Lambda Notation

Lambda notation is a syntax for defining function values directly rather than using `def`

```
lambda x1, x2,...: e end
```

Convenient for defining small functions to pass to map/filter/fold, etc.

Implementing First-Class Functions

How can we implement first class functions:

1. What is the runtime representation of a function value?

2. What data do we need to correctly implement **dynamic** type checking for functions

3. How can we ensure that we handle **variable capture** ?

Function Pointers

The compiled instructions implementing our functions are stored in executable memory.

The simplest way to implement functions as first class values is for the **runtime representation** of a function to simply be the **address of its code** in memory.

Function Pointers

Live code example

Function Pointers

The compiled instructions implementing our functions are stored in executable memory.

The simplest way to implement functions as first class values is for the **runtime representation** of a function to simply be the **address of its code** in memory.

This representation works in C, a **statically typed** language where functions are only defined at the **top level**, i.e., cannot capture any variables.

Our language is **dynamically typed** and has **local function definitions.**

Dynamic Typing for Function Values

What new kinds of errors can arise with first class function values?

Dynamic Typing for Function Values

```
def applyToFive(it):
  it(5)
in

def incr(x):
  x + 1
in

applyToFive(true)
```

runtime error: **true** is not a function

Dynamic Typing for Function Values

```
def applyToFive(f):
  f(5)
in

def add(x, y):
  x + y
in

applyToFive(add)
```

runtime error: add (defined at ...) expected 2 arguments but was applied to 1

Dynamic Typing for Function Values

- Need to tag function values so that we can distinguish them from other data.

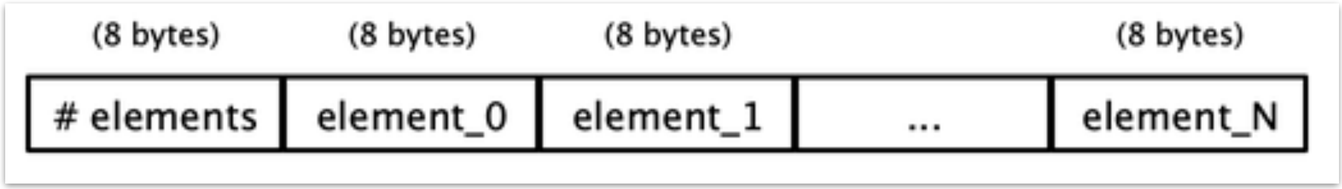
- Need to store the **arity**, i.e., number of parameters, with the function. Similar to storing array length.

Dynamically typed function pointer then needs to take up more than 8 bytes to store the function pointer and the arity, so we can store this as **boxed** data stored on the heap.

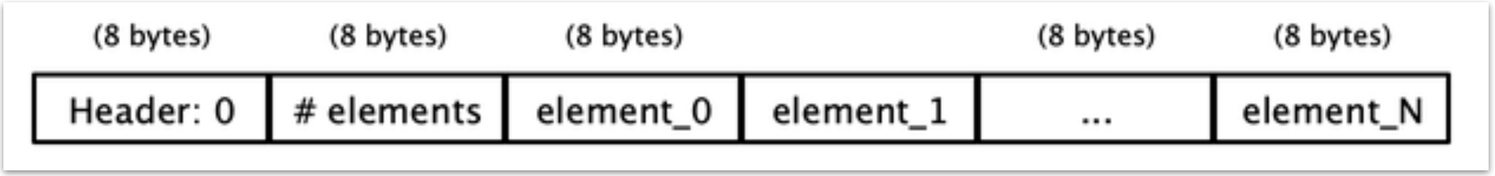
Heap Object Metadata

We have already used 2 bits in our Snake values for tagging datatypes. If values are **boxed** we can easily allocate many more by using a **header word** in our heap representation.

E.g., for arrays:



Update to include an initial 8 byte header that identifies the type of the object on the heap. For arrays: tag 0



Heap Object Metadata

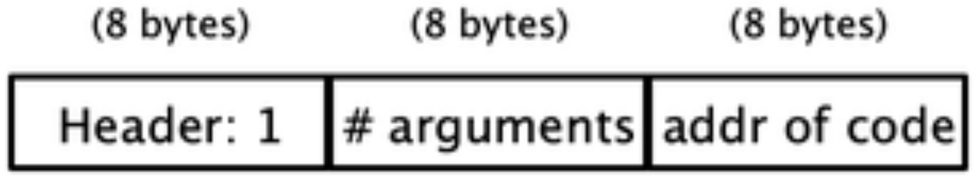
We have already used 2 bits in our Snake values for tagging datatypes. If values are **boxed** we can easily allocate many more by using a **header word** in our heap representation.

For function pointers, we need to store

- A different header tag

- the number of arguments

- the function pointer itself



Function Pointers

The compiled instructions implementing our functions are stored in executable memory.

The simplest way to implement functions as first class values is for the **runtime representation** of a function to simply be the **address of its code** in memory.

This representation works in C, a **statically typed** language where functions are only defined at the **top level**, i.e., cannot capture any variables.

Our language is **dynamically typed** and has **local function definitions.**

Variable Capture

```
let a = [0, 1, 2, 3] in
let _ map(lambda i: a[i] := a[i] * a[i] end, [1,3]) = in
a
```

the lambda function here **captures** the array variable **a**

Variable Capture

For second-class functions, we used lambda lifting to implement variable capture.

Key property of second-class functions: at a call site, we can statically determine the function we are being called with. So we can lookup what extra arguments the function requires

This property fails for first-class functions

Variable Capture

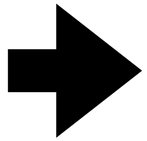
```
def adder(x):
  lambda y: x + y end
in
let add1 = adder(1),
    add2 = adder(2),
in add1(add2(0))
```

the lambda function here **captures** the outer variable **x**

what happens if we attempt our lambda lifting translation from before?

Variable Capture

```
def adder(x):
  lambda y: x + y end
in
let add1 = adder(1),
    add2 = adder(2),
in add1(add2(0))
```



```
fun adder(x):
  f = lambda_fun
  ret f
fun lambda_fun(x,y):
  r = x + y
  ret r
entry:
  add1 = adder(1)
  add2 = adder(2)
  tmp = add2(?, 0)
  tmp2 = add1(?, tmp)
  ret tmp2
```

at the **call site** we don't know the values of the captured variables

Variable Capture

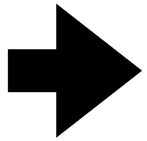
```
def main(b):
  let f =
    if b:
      let x = 7 in
      lambda y:
        x + y end
    else: lambda y: y end
  in
  f(5) + 1
```

the value of **f** is
dynamically determined

what happens if we attempt our lambda lifting translation from before?

Variable Capture

```
def main(b):
  let f =
    if b:
      let x = 7 in
      lambda y:
        x + y end
    else: lambda y: y end
  in
  f(5) + 1
```



```
fun lambda_1(x,y):
  r = x + y
  ret r
fun lambda_2(y):
  ret y
entry(b):
  jn(f):
    x = f(?, ?, 5) or f(?, 5)
    o = x + 1
    ret o
  thn:
    x = 7
    br jn(lambda_1)
  els:
    br jn(lambda_2


  cbr b thn() els()
```

at the **call** site, we don't know **how many** values are captured

Functions Pointers can't Capture

Just like second class functions, first-class functions can **capture** variables in scope at their definition site.

Unlike second class functions, the caller of a first-class function

1. Doesn't know how many variables the function captured

2. May not even have access to the variables the function captured

Our current strategy of supplying captured variables as extra arguments at the **call site** is doomed to fail for first-class functions that can capture.

Alternative: need to supply the captured variables at the **definition site.**

Local Functions Attempt 1: Runtime Code Generation

One strategy to implement local functions is **runtime code generation**.

- A function value at runtime is still a (tagged) function pointer

- Constructing a function value means compiling the code at runtime, when the values of captured variables are determined.

```
def adder(x):
  lambda y: x + y end
in
let add1 = adder(1),
    add2 = adder(2),
in add1(add2(0))
```

The first call to adder triggers a compilation of the code `lambda y: 1 + y`. Stores this in the heap and returns the pointer.

Local Functions Attempt 1: Runtime Code Generation

One strategy to implement local functions is **runtime code generation**.

- A function value at runtime is still a (tagged) function pointer

- Constructing a function value means compiling the code at runtime, when the values of captured variables are determined.

Advantage:

The generated code can be more efficient because the values are known at runtime.

Disadvantage:

Big runtime overhead to run the compiler down to binary at runtime.

Not common in ahead-of-time compilation, but similar to how Just-in-time compilers work

Local Functions Attempt 2: Closures

The most common strategy to implement local functions is to use **closures**.

- A function value at runtime is a heap-allocated object grouping a function pointer with an array containing the values of its captured variables

- Constructing a function value involves storing the captured variables on the heap

- To call a function, unpack its code pointer and pass a pointer to its captured variables.

Closure Conversion

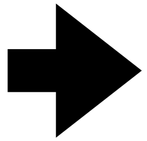
Similar to our lambda lifting pass, we can translate programs that implicitly use closures and capture variables to one that explicitly constructs/deconstructs them. This pass is called **closure conversion.**

Can be done at the AST level or the SSA level, just like lambda lifting

Closure Conversion

An easy way to implement closure conversion is to compile to a version of the language with arrays + function pointers.

```
def adder(x):
  lambda y: x + y end
in
let add1 = adder(1),
    add2 = adder(2),
in add1(add2(0))
```



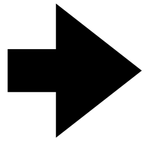
```
def lambda_fun(captures,y):
  captures[0] + y
in
def adder(x): [lambda_fun, [x]] in

let add1 = adder(1),
    add2 = adder(2),
in add1[0](add1[1],
     add2[0](add2[1], 0))
```

By storing the captured variables as part of the function value, we can supply them at the definition site, and use them at the call site

Closure Conversion

An easy way to implement closure conversion is to compile to a version of the language with arrays + function pointers.

```
def main(b):
  let f =
    if b:
      let x = 7 in
      lambda y:
        x + y end
    else: lambda y: y end
  in
  f(5) + 1
```



```
def lambda_1(captures, y):
  captures[0] + y
in
def lambda_2(captures, y): y in
def main(b):
  let f =
    if b: let x = 7 in [lambda_1, [x]]
    else: [lambda_2, []]
  in
  f[0](f[1], 5) + 1
```

Important to represent all lambda functions as taking an array of arguments so that they have a uniform interface: the caller doesn't know how large the captured environment is

Closure Conversion

Translating closures to arrays + function pointers gives correct semantics, but doesn't support dynamic typing features. E.g., all closures values would satisfy **isArray**.

Instead make a new type of heap object for closures

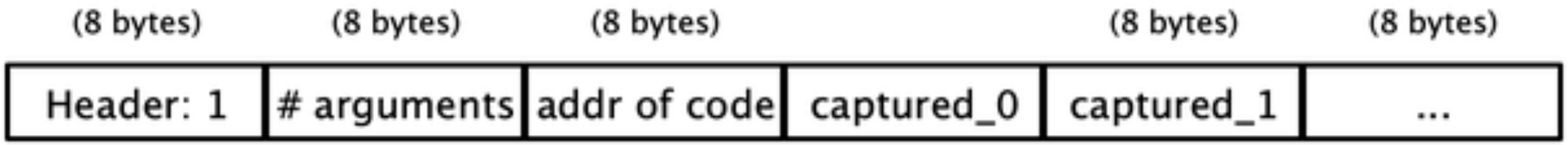
Functions as Closures

A **closure** is a datatype for first-class functions consisting of both

1. The function pointer

2. An array of **captured arguments**

We store the captured arguments at the function's **definition site**, rather than passing them at the **call site**



Recursive Closures

How can we extend this closure conversion strategy to handle recursive functions?

Surprisingly, we can also "translate away" recursive definitions, in two ways:

1. Clever functional programming: **Y combinator**

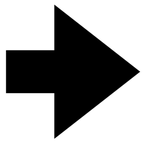
2. Clever imperative programming: "Landin's knot"

Recursion as Syntax Sugar

How do we "translate away" recursion?

Step 1: translate a recursive function into one that takes "itself" as an argument.

```
def fact(n):
  if n == 0: 1
  else: n * fact(n - 1)
```



```
let fact = lambda(fact): lambda(n):
  if n == 0: 1
  else: n * fact(n - 1)
  end
end
```

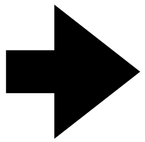
Recursion as Syntax Sugar

How do we "translate away" recursion?

Step 1: translate a recursive function into one that takes "itself" as an argument.

Step 2: write a function that "ties the knot" applying its input to itself

```
def fact(n):
  if n == 0: 1
  else: n * fact(n - 1)
```



```
let fact = Y(lambda(fact): lambda(n):
  if n == 0: 1
  else: n * fact(n - 1)
  end
end)
```

Y Combinator

The Y combinator is one method for "tying the knot". Discovered in the 1930s when lambda calculus was invented as a foundation for logic.

```
let Y = lambda f:
  let s = lambda x: lambda v:
            f(x(x))(v) end end
  in s(s)
  end
```

Y Combinator

Why does this work?

```
Y(f)
=~ let s = (lambda x: lambda v: f(x(x))(v) end end)
   in s(s)

=~ let s = (lambda x: lambda v: f(x(x))(v) end end)
   lambda v: f(s(s))(v)

=~ lambda v: f(Y(f))(v)
```

Y Combinator

Demo: JS Y Combinator

Y Combinator

The Y combinator is one method for "tying the knot". Discovered in the 1930s when lambda calculus was invented as a foundation for logic.

```
let Y = lambda f:
  let s = lambda x: lambda v:
            f(x(x))(v) end end
  in s(s)
  end
```

Elegant, but allocates a lot of closures.

Challenge: extend this to mutual recursion and functions with any number of arguments

Landin's Knot

An easier way to "tie the knot" is to "backpatch" a pointer to the function.

```
let factBox = [false] in
let fact = lambda n:
  if n == 0: 1
  else: n * factBox[0](n - 1)
  end
end)
```

Use array of length 1 as a way to get a mutable variable

Use false as a "null pointer"

Landin's Knot

Demo: JS Landin's Knot

Landin's Knot

An easier way to "tie the knot" is to "backpatch" a pointer to the function.

```
let factBox = [false] in
let fact = lambda n:
  if n == 0: 1
  else: n * factBox[0](n - 1)
  end
end)
```

In memory: we are constructing a circular data structure. Initialize the recursive references to null pointers and then update them once the data is defined.

Compiling Functions

In our source programming languages, functions are a simple, elegant abstraction.

But they do not have a single elegant implementation.

Modern compilers work hard to combine multiple implementation strategies behind this single source interface:

1. Local tail calls can be compiled as efficiently as loops

2. Most calls are to statically determined functions, don't require allocating a closure

3. Construct a closure only when necessary: when the function is actually used in a first class manner.

Essential in performance-sensitive languages like Rust that use closures for core functionality (iterators)!