FDL Closures

Free Variables and Lambdas

Free Variables of a `lambda`

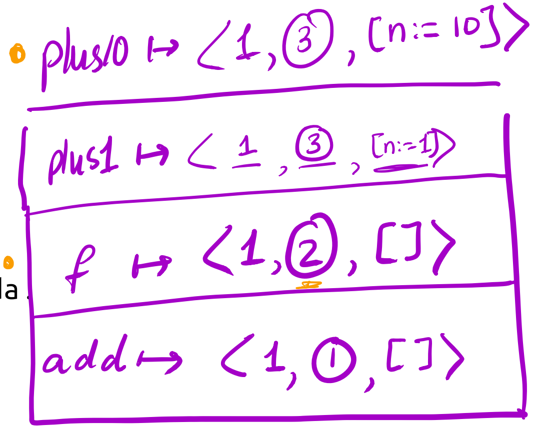
- Those whose values come from *outside*
- Should use *the same* values whenever we “call” the `lambda`.

For example:

```
let add    = (lambda (n): (lambda (m): n + m))
  , f      = (lambda (it): it(5))
  , plus1  = add(1)
  , plus10 = add(10)
in
  (f(plus1), f(plus10), plus10(20))
```

should evaluate to `(6, 15, 30)`

- `plus1` be like `lambda (m): 1 + m`
- `plus10` be like `lambda (m): 10 + m`



Quiz

"arity" of add is

(A) 1

(B) 2

Achieving Closure

(Recall from CSE 130)

```
let add    = (lambda (n): (lambda (m): n + m))
  , f      = (lambda (it): it(5))
  , plus1  = add(1)
  , plus10 = add(10)
in
  (f(plus1), f(plus10), plus10(20))
```

should evaluate to `(6, 15, 30)`

- `plus1` be like `lambda (m): 1 + m`
- `plus10` be like `lambda (m): 10 + m`

Key Idea: Each function value must **store its free variables**

represent `plus1` as:

```
(arity, code-label, [n := 1])
```

represent `plus10` as:

```
(arity, code-label, [n := 10])
```

Same code, but different free variables.

Lam xs e

App e [e_1, ..., e_n]

Strategy Progression

1. **Representation** = `Start-Label`
   - **Problem:** How to do run-time checks of valid args?
2. **Representation** = `(Arity, Start-Label)`
   - **Problem:** How to map function **names** to tuples?
3. **Lambda Terms** Make functions just another expression!
   - **Problem:** How to store local variables?
4. **Function Value** `(Arity, Start-Label, Free_1, ... , Free_N)`
   - **Ta Da!**

Closures: Strategy

What if we have *multiple* free variables?

```
let foo    = (lambda (x, y):
                (lambda (z): x + y + z)
             )
  , plus10 = foo(4, 6)
  , plus20 = foo(7, 13)
in
  (plus10(0), plus20(100))
```

represent `plus10` as:

```
(arity, code-label, [x := 4], [y := 6])
```

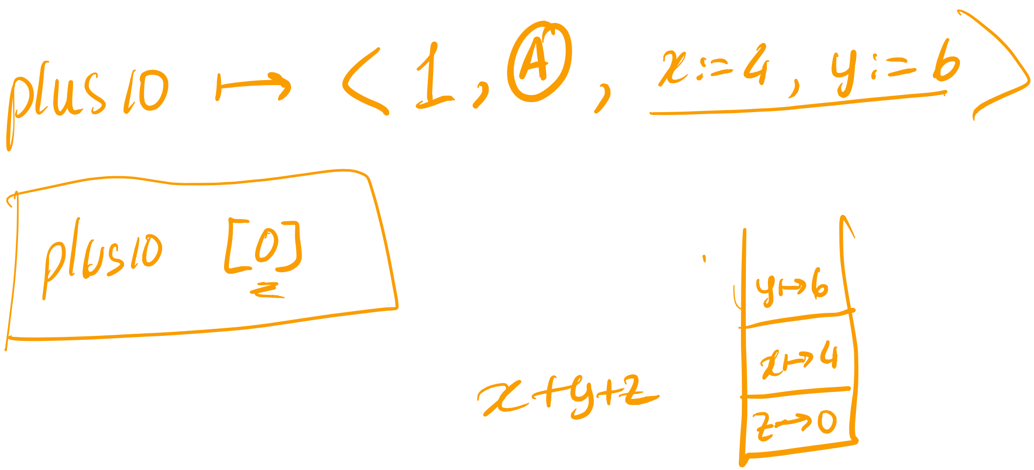
represent `plus20` as:

```
(arity, code-label, [x := 7], [y := 13])
```

Example

Lets see how to evaluate

```
let foo    = (lambda (x, y):
               (lambda (z): x + y + z)
             )
  , plus10 = foo(4, 6)
in
  plus10(0)
```



Example

Lets see how to evaluate

```
let foo    = (lambda (x, y):
                lambda (z): x + y + z)
              )
  , plus10 = foo(4, 6)
  , f      = (lambda (it): it(5))
in
  f(plus10)
```

Implementation

Representation

1. How to store closures

Types:

- Same as before

Transforms

1. Update `tag` and `ANF`
   - as before
2. Update `checker`
3. Update `compile`

Representation

We can represent a **closure** as a tuple of

```
(arity, code-ptr, free-var-1, ... free-var-N)
```

which means, following the convention for tuples, as:

```
-------------------------------------------------
| N + 2 | arity | code-ptr | var1 | ... | varN |
-------------------------------------------------
```

Where each cell represents 64-bits / 8-bytes / 1-(double)word.

Note: (As with all tuples) the first word contains the #elements of the tuple.

- In this case `N + 2`

Transforms: Checker

What environment should we use to check a `Lam` **body** ?

```
wellFormed :: BareExpr -> [UserError]
wellFormed = go emptyEnv
  where
    ...
    go vEnv (Lam xs e _) = errDupParams xs
                        ++ go ?vEnv e

addsEnv :: Env -> [BareBind] -> Env
addsEnv env xs = foldr addEnv env xs
```

QUIZ How shall we implement `?vEnv` ?

A. `addsEnv vEnv []`

B. `addsEnv vEnv xs`

C. `addsEnv emptyEnv xs`

Transforms: Compile

Question How does the called function **know** the values of free vars?

RESTORE them from Clos → STACK

- Needs to **restore them** from closure tuple

- Needs to **access** the closure tuple!

... But how shall we give the called function **access** to the tuple?

By passing the tuple as an *extra parameter*

Transforms: Compile

Calls `App` (as before)

App e [e1, e2, ... en]

0. eval e into RAX

1. **Push** *closure-pointer* + parameters
2. **Call** code-label
3. **Pop** *closure-pointer* + params

e[1]   "real" params

check arity

jump label_end
labelstart:
  <code for body>   restore FV
label end:
mktuple (len xs, labelStart, <y1>, <y2>, ...)
set tag

arity   codeptr

(Lam xs e)

ys = free(lam xs e)

y1, y2 ...

Definitions `Lam`

1. **Compute** *free-vars* `x1` ,..., `xn`
2. **Generate** `code-block`

- **Restore** free vars from closure-pointer-parameter **New**
- **Execute** function body (as before)

3. **Allocate** tuple `(arity, code-label, x1, ... , xn)`

Transforms: Compile Definitions

1. **Compute** *free-vars* `y1` ,..., `yn`
2. **Generate** code-block

- **Restore** free vars from closure-pointer-parameter
- **Execute** function body (as before)

3. **Allocate** tuple `(arity, code-label, y1, ... , yn)`

```
compileEnv :: Env -> AExp -> [Instruction]
compileEnv env (Lam xs e l)
  = IJmp   end                          -- Why?
  : ILabel start                        -- Function start
  : lambdaBody ys xs e                  -- Function code (like Decl)
 ++ ILabel end                          -- Function end
  : lamTuple arity start env ys         -- Compile closure-tuple into RA

  where
    ys    = freeVars (Lam xs e l)
    arity = length xs
    start = LamStart l
    end   = LamEnd   l
```

Creating Closure Tuples

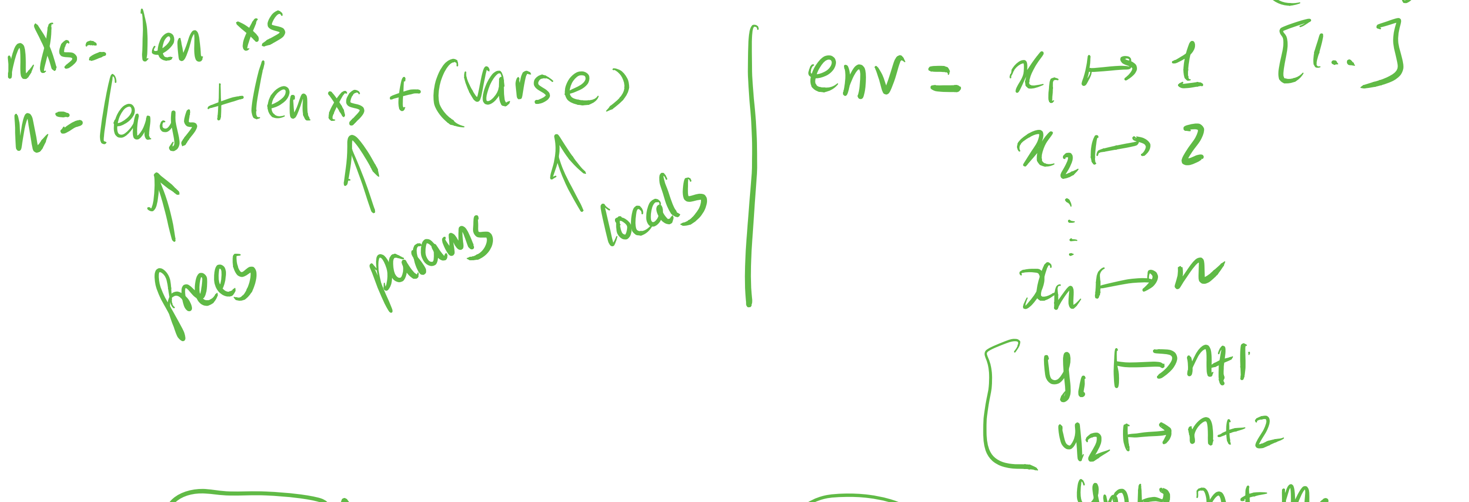
To create the actual closure-tuple we need

- the **free-variables** `ys`
- the `env` from which to **values** of the free variables.

```
lamTuple :: Int -> Label -> Env -> [Id] -> [Instruction]
lamTuple arity start env ys
  =  tupleAlloc  (2 + length ys)                     -- alloc tuple 2
+ |ys|
  ++ tupleWrites ( repr arity                        -- fill arity
                 : CodePtr start                     -- fill code-ptr
                 : [immArg env (Id y) | y <- ys] )   -- fill free-vars
  ++ [ IOr  (Reg RAX) (typeTag TClosure) ]           -- set the tag bi
ts
```

Generating Code Block

```
lambdaBody :: [Id] -> [Id] -> AExp -> [Instruction]
lambdaBody ys xs e =
    funEntry n                  -- 1. setup  stack frame RBP/RSP
 ++ copyArgs xs'                -- 2. copy parameters to stack slots
 ++ restore nXs ys              -- 3. copy (closure) free vars to stack
slots
 ++ compileEnv env body         -- 4. execute 'body' with result in RAX
 ++ funExit n                   -- 5. teardown stack frame & return
```



To `restore ys` we use the closure-ptr passed in at `[RDI]` – the special **first** parameter – to copy the free-vars `ys` onto the stack.

```
restore :: Int -> [Id] -> [Instruction]
restore base ys =
  concat [ copy i | (_, i) <- zip ys [1..]]
    where
      closV  = Reg RDI
      copy i = tupleReadRaw closV (repr (i+1))        -- copy tuple-
fld for y into RAX...
            ++ [ IMov (stackVar (base+i)) (Reg RAX) ]  -- ...write RA
X into stackVar for y
```

A Problem: Recursion

Oops, how do we write:

```
def fac(n):
  if (n > 1):
    n * fac(n-1)
  else:
    1

fac(5)
```

If we try

```
let fac = (lambda (n):
             if (n < 1):
               1
             else:
               n * fac(n-1))
in fac(5)
```

We get a variable unbound error!

```
Errors found!
tests/input/fac-bad.fdl:5:20-23: Unbound variable 'fac'

         5|                   n * fac(n-1))
```

We need to teach our compiler that its ok to use the name `fac` inside the body!

Solution: Named Functions

We have a new form of **named functions** which looks like this:

```
def fac(n):
  if (n < 1):
    1
  else:
    n * fac(n - 1)
in
  fac(5)
```

Representing Named Functions

We extend `Expr` to handle such functions as:

```
data Expr a
  = ...
  | Fun     (Bind a)        -- ^ name of function
            [Bind a]        -- ^ list of parameters
            (Expr a) a      -- ^ body of function
```

Note that we parse the code

```
def foo(x1,...,xn):
  e
in
  e'
```

as the `Expr`

```
Let foo (Fun foo [x1,...,xn] e) e'
```

env

env? = [foo, x_1, ..., x_n]

foo is allowed to appear in e

Compiling Named Functions

Mostly, this is left as an exercise to *you*

Non-Recursive functions

- i.e. `f` *does not* appear inside `e` in `Fun f xs e`
- Treat `Fun f xs e` as `Lam xs e` …
- … Everything should *just work.*

Recursive

- i.e. `f` *does* appear inside `e` in `Fun f xs e`
- Can you think of a simple tweak to the `Lam` strategy that works?

Recap: Functions as Values

We had functions, but they were *second-class* entities…

Now, they are *first-class* values

- passed around as parameters
- returned from functions
- stored in tuples etc.

How?

1. **Representation** = `Start-Label`
   - **Problem:** How to do run-time checks of valid args?
2. **Representation** = `(Arity, Start-Label)`
   - **Problem:** How to map function **names** to tuples?
3. **Lambda Terms** Make functions just another expression!
   - **Problem:** How to store local variables?
4. **Function Value** `(Arity, Start-Label, Free_1, ... , Free_N)`
   - **Ta Da!**

Next: Adding **garbage collection**

- *Reclaim!* Heap memory that is no longer in use

Next: Adding **static type inference**

- *Faster!* Gets rid of those annoying (and slow!) run-time checks
- *Safer!* Catches problems at compile-time, when easiest to fix!

(https://ucsd-cse131.github.io/sp21/feed.xml)

(https://twitter.com/ranjitjhala)

● (https://plus.google.com/u/0/106612421534244742464)

● (https://github.com/ucsd-cse131/sp21)