Nano: Functions

Let’s add

- **lambda abstraction** (aka function definitions)
- **application** (aka function calls)

\[
e ::= n \quad \text{-- OLD}
| \text{e}_1 \ `\text{op}` \ e_2
| x
| \text{let } x = \text{e}_1 \text{ in } \text{e}_2
| \lambda x -> \text{e} \quad \text{-- abstraction}
| \text{e}_1 \text{ e}_2 \quad \text{-- application}
\]

Example

\[
\text{let } \text{incr} = \lambda x -> x + 1 \\
\text{in} \\
\text{incr } 10
\]

Representation
Representation

data Expr
  = ENum Int -- OLD
  | EBin Binop Expr Expr
  | EVar Id
  | ELet Id Expr Expr -- NEW
  | ??? -- abstraction \x -> e
  | ??? -- application (e1 e2)

Example

let incr = \x -> x + 1
in
  incr 10

is represented as
Functions are Values

Recall the trinity

But... what is the value of a function?

Let's build some intuition with examples.

**QUIZ**
What does the following expression evaluate to?

```
let incr = \x -> x + 1 -- abstraction ("definition")
in incr 10 -- application ("call")
```

(A) Error/Undefined
(B) 10
(C) 11
(D) 0
(E) 1

What is the Value of `incr`?

- Is it an Int?
- Is it a Bool?
- Is it a ???

What information do we need to store (in the Env) about `incr`?
A Function’s Value is its Code

```
let incr = \x -> x + 1
```

```
-- env
```

```
let "incr" := <code> : env
```

```
incr 10
```

-- evaluate <code> with parameter := 10

What information do we store about <code>?

A Call’s Value

How to evaluate the “call” incr 10?

1. Lookup the <code> i.e. <param, body> for incr (stored in the environment),
2. Evaluate \textbf{body} with \texttt{param} set to \texttt{10}!

\textbf{Two kinds of Values}

We now have two kinds of Values

\[ v ::= n \quad \text{--- OLD} \]
\[ \mid <x, e> \quad \text{--- <param, body>} \]

1. Plain \texttt{Int} (as before)
2. A function’s “code”: a pair of “parameter” and “body-expression”

\texttt{data Value} \hspace{1cm} \texttt{= VInt \hspace{0.4cm} Int \hspace{0.2cm} \text{--- OLD}}
\[\mid \text{VCode \hspace{0.3cm} Id \hspace{0.1cm} Expr \hspace{0.2cm} \text{--- <x, e>}}\]

\textbf{Evaluating Lambdas and Applications}
eval :: Env -> Expr -> Value

-- OLD

eval env (ENum n) = ???
eval env (EVar x) = ???
eval env (EBin op e1 e2) = ???
eval env (ELet x e1 e2) = ???

-- NEW

eval env (ELam x e) = ???
eval env (EApp e1 e2) = ???

Lets make sure our tests work properly!

exLam1 = ELet "incr" (ELam "x" (EBin Add (EVar "x") (ENum 1)))
          (EApp (EVar "incr") (ENum 10))

-- >>> eval [] exLam1
-- 11

QUIZ

What should the following evaluate to?

let c = 1
in
let inc = \x -> x + c
in
inc 10

(A) Error/Undefined
(B) 10
exLam2 = ELet "c" (ENum 1)
   (ELet "incr" (ELam "x" (EBin Add (EVar "x") (EVar "c")))))
   (EApp (EVar "incr") (ENum 10))

-- >>> eval [] exLam2
-- ????

**QUIZ**

And what should this expression evaluate to?
let $c = 1$

in

\[
\text{let inc = } \lambda x \to x + c
\]

in

\[
\text{let } c = 100
\]

in

\[
\text{inc 10}
\]

(A) Error/Undefined

(B) 110

(C) 11

**The “Immutability Principle”**

A function’s behavior should *never change*

- A function must *always* return the same output for a given input

Why?
> myFunc 10
0

> myFunc 10
10

Oh no! How to find the bug? Is it

- In myFunc or
- In a global variable or
- In a library somewhere else or
- ...

My worst debugging nightmare

Colbert “Immutability Principle” (https://youtu.be/CWqzLgDc030?t=628)

**The Immutability Principle?**

How does our `eval` work?

```plaintext
exLam3 = ELet "c" (ENum 1)
    (ELet "incr" (ELam "x" (EBin Add (EVar "x") (EVar "c")))
        (ELet "c" (ENum 100)
            (EApp (EVar "incr") (ENum 10)
                ))
    )

-- >>> eval [] exLam3
-- ???
```
Oops?

```haskell
let c = 1
in
  let inc = \x -> x + c
  in
    let c = 100
    in
      = 1] <<< env
      inc 10
```

And so we get

```haskell
eval env (inc 10)

===> eval ("x" := 10 : env) (x + c)

===> 10 + 100

===> 110
```

Ouch.

**Enforcing Immutability with Closures**

How to enforce immutability principle

- inc 10 always returns 11?

**Key Idea: Closures**
At definition: Freeze the environment the function’s value

At call: Use the frozen environment to evaluate the body

Ensures that inc 10 always evaluates to the same result!

```
let c = 1
in
  let inc = \x -> x + c
  in
    -- ["c" := 1] <<< frozenv = ["c" := 1]

let c = 100
in
  -- ["c" := 100, "inc" := <frozenv, x, x+c>, c := 1] <<< frozenv = ["c" := 1]
  inc 10
```

Now we evaluate

eval env (inc 10)

```
  ==> eval ("x" := 10 : frozenv) (x + c) where frozenv = ["c" := 1]

  ==> 10 + 1

  ==> 1
```

tada!

**Representing Closures**

Lets change the Value datatype to also store an Env
data Value
  = VInt Int -- OLD
  | VClos Env Id Expr -- <frozenv, param, body>

Evaluating Function Definitions

How should we fix the definition of eval for ELam?

eval :: Env -> Expr -> Value

eval env (ELam x e) = ???

Hint: What value should we bind incr to in our example above?

(Recall At definition freeze the environment the function’s value)

Evaluating Function Calls

How should we fix the definition of eval for EApp?

eval :: Env -> Expr -> Value

eval env (EApp e1 e2) = ???

(Recall At call: Use the frozen environment to evaluate the body)

Hint: What value should we evaluate incr 10 to?
Let’s generalize that recipe!

1. Evaluate $\text{e1}$ to get $\langle \text{frozen}, \text{param}, \text{body} \rangle$
2. Evaluate $\text{e2}$ to get $\text{v2}$
3. Evaluate $\text{body}$ in $\text{param} := \text{v2} : \text{frozen}$

**Immutability Achieved**

Let's put our code to the test!

```
exLam3 = ELet "c" (Enum 1)  
    (ELet "incr" (ELam "x" (EBin Add (EVar "x") (EVar "c")))  
        (ELet "c" (Enum 100)  
            (EApp (EVar "incr") (Enum 10)  
            )))  
    )
```

```
-- >>> eval [] exLam3
-- ???
```

**QUIZ**

What should the following evaluate to?
let add = \x -> (\y -> x + y)
in
let add10 = add 10
in
let add20 = add 20
in
(let add10 = add 10
let add20 = add 20

\( \text{add10 100} \) + \( \text{add20 1000} \) + 1130

\( \text{add 10} 100 \)

Functions Returning Functions Achieved!

exLam4 = ...

-- >>> eval [] exLam4

Functions Accepting Functions Achieved!

Practice

What should the following evaluate to?

let add = \x -> (\y -> x + y)
in
let add10 = add 10
in
let doTwice = \f -> (\x -> f (f x))
in
doTwice add10 100

eval [120]
The Nano Language

Features of Nano:

1. Arithmetic expressions [done]
2. Variables [done]
3. Let-bindings [done]
4. Functions [done]
5. Recursion

... You figure it out Hw4 ... :-)