CMPS 112: Spring 2019

Comparative Programming Languages

Environments and closures

Owen Arden
UC Santa Cruz

Based on course materials developed by Nadia Polikarpova

Roadmap

Past three weeks:

- How do we *use* a functional language?

Next three weeks:

- How do we *implement* a functional language?
- … in a functional language (of course)

This week: Interpreter

- How do we *evaluate* a program given its abstract syntax tree (AST)?
- How do we *prove properties* about our interpreter (e.g. that certain programs never crash)?

The Nano Language

Features of Nano:

1. Arithmetic expressions
2. Variables and let-bindings
3. Functions
4. Recursion

Reminder: Calculator

Arithmetic expressions:

```
e ::= n
    | e1 + e2
    | e1 - e2
    | e1 * e2
```

Example:

```
4 + 13
==> 17
```


Reminder: Calculator

Haskell datatype to *represent* arithmetic expressions:

```
data Expr = Num Int
          | Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
```

Haskell function to *evaluate* an expression:

```
eval :: Expr -> Int
eval (Num n)     = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Sub e1 e2) = eval e1 - eval e2
eval (Mul e1 e2) = eval e1 * eval e2
```


Reminder: Calculator

Alternative representation:

```
data Binop = Add | Sub | Mul

data Expr = Num Int                -- number
          | Bin Binop Expr Expr    -- binary expression
```

Evaluator for alternative representation:

```
eval :: Expr -> Int
eval (Num n)         = n
eval (Bin Add e1 e2) = eval e1 + eval e2
eval (Bin Sub e1 e2) = eval e1 - eval e2
eval (Bin Mul e1 e2) = eval e1 * eval e2
```

The Nano Language

Features of Nano:

1. Arithmetic expressions **[done]**
2. Variables and let-bindings
3. Functions
4. Recursion

Extension: variables

Let's add variables and `let` bindings!

```
e ::= n | x
    | e1 + e2 | e1 - e2 | e1 * e2
    | let x = e1 in e2
```

Example:

```
let x = 4 + 13 in   -- 17
let y = 7 - 5 in    -- 2
x * y

==> 34
```


Extension: variables

Haskell representation:

```
data Expr = Num Int               -- number
          | ???                   -- variable
          | Bin Binop Expr Expr   -- binary expression
          | ???                   -- let expression
```

Extension: variables

```haskell
type Id = String

data Expr = Num Int               -- number
          | Var Id                -- variable
          | Bin Binop Expr Expr   -- binary expression
          | Let Id Expr Expr      -- let expression
```

Haskell function to *evaluate* an expression:

```haskell
eval :: Expr -> Int
eval (Num n)         = n
eval (Var x)         = ???
...
```


Extension: variables

```haskell
type Id = String

data Expr = Num Int               -- number
```

How do we evaluate a variable?

We have to remember
which *value* it was bound to!

Haskell function

```haskell
eval (Var x)         = ???
...
```


Environment

An expression is evaluated in an **environment**, which maps all its *free variables* to *values*

Examples:

```
x * y
=[x:17, y:2]=> 34
```

- How should we represent the environment?
- Which operations does it support?

```
x * y
=[x:17]=> Error: unbound variable y

x * (let y = 2 in y)
=[x:17]=> 34
```

Extension: variables

What does this evaluate to? *

```
let x = 5 in
let y = x + z in
let z = 10 in
y
```

- (A) 15
- (B) 5
- (C) Error: unbound variable x
- (D) Error: unbound variable y
- (E) Error: unbound variable z

http://tiny.cc/cmps112-vars-ind

Extension: variables

What does this evaluate to? *

```
let x = 5 in
let y = x + z in
let z = 10 in
y
```

- (A) 15
- (B) 5
- (C) Error: unbound variable x
- (D) Error: unbound variable y
- (E) Error: unbound variable z

http://tiny.cc/cmps112-vars-grp

Environment: API

To evaluate `let x = e1 in e2` in `env`:

- evaluate `e2` in an **extended** environment `env + [x:v]`
- where `v` is the result of evaluating `e1`

To evaluate `x` in `env`:

- **lookup** the most recently added binding for `x`

```
type Value = Int

data Env = ... -- representation not that important

-- | Add a new binding
add    :: Id -> Value -> Env -> Env

-- | Lookup the most recently added binding
lookup :: Id -> Env -> Value
```

Evaluating expressions

Back to our expressions... now with environments!

```
data Expr = Num Int              -- number
          | Var Id               -- variable
          | Bin Binop Expr Expr  -- binary expression
          | Let Id Expr Expr     -- let expression
```


Evaluating expressions

Haskell function to *evaluate* an expression:

```
eval :: Env -> Expr -> Value
eval env (Num n)         = n
eval env (Var x)         = lookup x env
eval env (Bin op e1 e2)  = f v1 v2
  where
    v1 = eval env e1
    v2 = eval env e2
    f = case op of
          Add -> (+)
          Sub -> (-)
          Mul -> (*)
eval env (Let x e1 e2)   = eval env' e2
  where
    v    = eval env e1
    env' = add x v env
```


Example evaluation

Nano expression

```
let x = 1 in
let y = (let x = 2 in x) + x in
let x = 3 in
x + y
```

is represented in Haskell as:

```
exp1 =    Let "x"
            (Num 1)                       exp2
            (Let "y"
              (Add                      exp3
                (Let "x" (Num 2) (Var x))
         exp4   (Var x))
              (Let "x"               exp5
                (Num 3)
                (Add (Var x) (Var y))))
```

Example evaluation

```
eval [] exp1
=> eval []                      (Let "x" (Num 1) exp2)
=> eval [("x",eval [] (Num 1))] exp2
=> eval [("x",1)]
    (Let "y" (Add exp3 exp4) exp5)
=> eval [("y",(eval [("x",1)] (Add exp3 exp4))), ("x",1)]
    exp5
=> eval [("y",(eval [("x",1)] (Let "x" (Num 2) (Var "x"))
             + eval [("x",1)] (Var "x"))), ("x",1)]
    exp5
=> eval [("y",(eval [("x",2), ("x",1)] (Var "x") -- new binding for x
             + 1)), ("x",1)]
    exp5
=> eval [("y",(2  -- use latest binding for x
             + 1)), ("x",1)]
    exp5
=> eval [("y",3), ("x",1)]
    (Let "x" (Num 3) (Add (Var "x") (Var "y")))
```


Example evaluation

```
=> eval [("y",3), ("x",1)]
    (Let "x" (Num 3) (Add (Var "x") (Var "y")))
=> eval [("x",3), ("y",3), ("x",1)]              -- new binding for x
    (Add (Var "x") (Var "y"))
=> eval [("x",3), ("y",3), ("x",1)] (Var "x")
 + eval [("x",3), ("y",3), ("x",1)] (Var "y")
=> 3 + 3
=> 6
```


Example evaluation

Same evaluation in a simplified format (Haskell `Expr` terms replaced by their "pretty-printed version"):

```
   eval []
   {let x = 1 in let y = (let x = 2 in x) + x in let x = 3 in x + y}
=> eval [x:(eval [] 1)]
                {let y = (let x = 2 in x) + x in let x = 3 in x + y}
=> eval [x:1]
                {let y = (let x = 2 in x) + x in let x = 3 in x + y}
=> eval [y:(eval [x:1] {(let x = 2 in x) + x}), x:1]
                                        {let x = 3 in x + y}
=> eval [y:(eval [x:1]  {let x = 2 in x} + eval [x:1] {x}), x:1]
                                        {let x = 3 in x + y}
         -- new binding for x:
=> eval [y:(eval [x:2,x:1] {x}          + eval [x:1] {x}), x:1]
                                        {let x = 3 in x + y}
    -- use latest binding for x:
=> eval [y:(                2           + eval [x:1] {x}), x:1]
                                        {let x = 3 in x + y}
=> eval [y:(                2           + 1)             , x:1]
                                        {let x = 3 in x + y}
```

Example evaluation

```
=> eval [y:(                2           + 1)             , x:1]
                                             {let x = 3 in x + y}
=> eval [y:3, x:1]
                                             {let x = 3 in x + y}
   -- new binding for x:
=> eval [x:3, y:3, x:1]
                                                          {x + y}
=> eval [x:3, y:3, x:1] x + eval [x:3, y:3, x:1] y
   -- use latest binding for x:
=> 3 + 3
=> 6
```


Runtime errors

Haskell function to *evaluate* an expression:

```
eval :: Env -> Expr -> Value
eval env (Num n)         = n
eval env (Var x)         = lookup x env -- can fail!
eval env (Bin op e1 e2)  = f v1 v2
  where
    v1 = eval env e1
    v2 = eval env e2
    f = case op of
          Add -> (+)
          Sub -> (-)
          Mul -> (*)
eval env (Let x e1 e2)   = eval env' e2
  where
    v    = eval env e1
    env' = add x v env
```

How do we make sure `lookup` doesn't cause a run-time error?

Free vs bound variables

In `eval env e`, `env` must contain bindings for *all free variables* of `e`!

- an occurrence of `x` is **free** if it is not **bound**
- an occurrence of `x` is **bound** if it's inside `e2` where `let x = e1 in e2`
- evaluation succeeds when an expression is **closed**!

QUIZ

Which variables are free in the expression? *

```
let y = (let x = 2 in x) + x in
let x = 3 in
x + y
```

- (A) None
- (B) x
- (C) y
- (D) x y

http://tiny.cc/cmps112-free-ind

QUIZ

Which variables are free in the expression? *

```
let y = (let x = 2 in x) + x in
let x = 3 in
x + y
```

- (A) None
- (B) x
- (C) y
- (D) x y

http://tiny.cc/cmps112-free-grp

The Nano Language

Features of Nano:

1. Arithmetic expressions **[done]**
2. Variables and let-bindings **[done]**
3. Functions
4. Recursion

Extension: functions

Let's add lambda abstraction and function application!

```
e ::= n | x
    | e1 + e2 | e1 - e2 | e1 * e2
    | let x = e1 in e2
    | \x -> e   -- abstraction
    | e1 e2     -- application
```

Example:

```
let c = 42 in
let cTimes = \x -> c * x in
cTimes 2
==> 84
```


Extension: functions

Haskell representation:

```
data Expr = Num Int                 -- number
          | Var Id                  -- variable
          | Bin Binop Expr Expr     -- binary expression
          | Let Id Expr Expr        -- let expression
          | ???                     -- abstraction
          | ???                     -- application
```


Extension: functions

Haskell representation:

```
data Expr = Num Int                 -- number
          | Var Id                  -- variable
          | Bin Binop Expr Expr     -- binary expression
          | Let Id Expr Expr        -- let expression
          | Lam Id Expr             -- abstraction
          | App Expr Expr           -- application
```

Extension: functions

Example:

```
let c = 42 in
let cTimes = \x -> c * x in
cTimes 2
```

represented as:

```
Let "c"
  (Num 42)
  (Let "cTimes"
    (Lam "x" (Mul (Var "c") (Var "x")))
    (App (Var "cTimes") (Num 2)))
```


Extension: functions

Example:

```
let c = 42 in
let cTimes = \x -> c * x in
cTimes 2
```

How should we evaluate this expression?

```
   eval []
    {let c = 42 in let cTimes = \x -> c * x in cTimes 2}
=> eval [c:42]
                  {let cTimes = \x -> c * x in cTimes 2}
=> eval [cTimes:???, c:42]
                                             {cTimes 2}
```

What is the **value** of `cTimes`???

Rethinking our values

Until now: a program *evaluates* to an integer (or fails)

```
type Value = Int

type Env = [(Id, Value)]

eval :: Env -> Expr -> Value
```

Rethinking our values

What do these programs evaluate to?

```
(1)
\x -> 2 * x
==> ???

(2)
let f = \x -> \y -> 2 * (x + y) in
f 5
==> ???
```

Conceptually, (1) evaluates to itself (not exactly, see later). while (2) evaluates to something equivalent to `\y -> 2 * (5 + y)`

Rethinking our values

Now: a program evaluates to an integer or *a lambda abstraction* (or fails)

- Remember: functions are *first-class* values

Let's change our definition of values!

```
data Value = VNum Int
           | VLam ??? -- What info do we need to store?

-- Other types stay the same
type Env = [(Id, Value)]

eval :: Env -> Expr -> Value
```


Function values

How should we represent a function value?

```
let c = 42 in
let cTimes = \x -> c * x in
cTimes 2
```

We need to store enough information about `cTimes` so that we can later evaluate any *application* of `cTimes` (like `cTimes 2`)!

First attempt:

```
data Value = VNum Int
           | VLam Id Expr -- formal + body
```

Function values

Let's try this!

```
   eval []
    {let c = 42 in let cTimes = \x -> c * x in cTimes 2}
=> eval [c:42]
                  {let cTimes = \x -> c * x in cTimes 2}
=> eval [cTimes:(\x -> c*x), c:42]
                                            {cTimes 2}
    -- evaluate the function:
=> eval [cTimes:(\x -> c*x), c:42]
                                     {(\x -> c * x) 2}
    -- evaluate the argument, bind to x, evaluate body:
=> eval [x:2, cTimes:(\x -> c*x), c:42]
                                            {c * x}
=>                                          42 * 2
=>                                          84
```

Looks good… can you spot a problem?

QUIZ

What should this evaluate to? *

```
let c = 42 in
let cTimes = \x -> c * x in -- but which c???
let c = 5 in
cTimes 2
```

- (A) 84
- (B) 10
- (C) Error: multiple definitions of c

http://tiny.cc/cmps112-cscope-ind

QUIZ

What should this evaluate to? *

```
let c = 42 in
let cTimes = \x -> c * x in -- but which c???
let c = 5 in
cTimes 2
```

- (A) 84
- (B) 10
- (C) Error: multiple definitions of c

http://tiny.cc/cmps112-cscope-grp

Static vs Dynamic Scoping

What we want:

```
let c = 42 in
let cTimes = \x -> c * x in
let c = 5 in
cTimes 2
=> 84
```

Lexical (or **static**) scoping:

- each occurrence of a variable refers to the most recent binding *in the program text*
- definition of each variable is unique and known *statically*
- good for readability and debugging: don't have to figure out where a variable got "assigned"

Static vs Dynamic Scoping

What we **don't** want:

```
let c = 42 in
let cTimes = \x -> c * x in
let c = 5 in
cTimes 2
=> 10
```

Dynamic scoping:

- each occurrence of a variable refers to the most recent binding *during program execution*
- can't tell where a variable is defined just by looking at the function body
- nightmare for readability and debugging:

Static vs Dynamic Scoping

Dynamic scoping:

- each occurrence of a variable refers to the most recent binding *during program execution*
- can't tell where a variable is defined just by looking at the function body
- nightmare for readability and debugging:

```
let cTimes = \x -> c * x in
let c = 5 in
let res1 = cTimes 2 in -- ==> 10
let c = 10 in
let res2 = cTimes 2 in -- ==> 20!!!
res2 - res1
```

Function values

```
data Value = VNum Int
           | VLam Id Expr -- formal + body
```

This representation can only implement dynamic scoping!

```
let c = 42 in
let cTimes = \x -> c * x in
let c = 5 in
cTimes 2
```

evaluates as:

```
   eval []
   {let c = 42 in let cTimes = \x -> c * x in let c = 5 in cTimes 2}
```


Function values

```
   eval []
   {let c = 42 in let cTimes = \x -> c * x in let c = 5 in cTimes 2}
=> eval [c:42]
                 {let cTimes = \x -> c * x in let c = 5 in cTimes 2}
=> eval [cTimes:(\x -> c*x), c:42]
                                        {let c = 5 in cTimes 2}
=> eval [c:5, cTimes:(\x -> c*x), c:42]
                                                     {cTimes 2}
=> eval [c:5, cTimes:(\x -> c*x), c:42]
                                               {(\x -> c * x) 2}
=> eval [x:2, c:5, cTimes:(\x -> c*x), c:42]
                                                     {c * x}
  -- latest binding for c is 5!
=>                                                    5 * 2
=>                                                    10
```

Lesson learned: need to remember what `c` was bound to when `cTimes` was defined!

- i.e. "freeze" the environment at function definition

Closures

To implement lexical scoping, we will represent function values as *closures*

Closure = *lambda abstraction* (formal + body) + *environment* at function definition

```
data Value = VNum Int
           | VClos Env Id Expr -- env + formal + body
```

Closures

Our example:

```
   eval []
   {let c = 42 in let cTimes = \x -> c * x in let c = 5 in cTimes 2}
=> eval [c:42]
                 {let cTimes = \x -> c * x in let c = 5 in cTimes 2}
   -- remember current env:
=> eval [cTimes:<[c:42], \x -> c*x>, c:42]
                                             {let c = 5 in cTimes 2}
=> eval [c:5, cTimes:<[c:42], \x -> c*x>, c:42]
                                                          {cTimes 2}
=> eval [c:5, cTimes:<[c:42], \x -> c*x>, c:42]
                                         {<[c:42], \x -> c * x> 2}
  -- restore env to the one inside the closure, then bind 2 to x:
=> eval [x:2, c:42]
                                                          {c * x}
=>                                                        42 * 2
=>                                                        84
```


QUIZ

Which variables should be saved in the closure environment of f? *

```
let a = 20 in
let f =
  \x -> let y = x + 1 in
        let g = \z -> y + z in
        a + g x
  in ...
```

- (A) a
- (B) a x
- (C) y g
- (D) a y g
- (E) a x y g z

http://tiny.cc/cmps112-env-ind

QUIZ

Which variables should be saved in the closure environment of f? *

```
let a = 20 in
let f =
  \x -> let y = x + 1 in
        let g = \z -> y + z in
        a + g x
  in ...
```

- (A) a
- (B) a x
- (C) y g
- (D) a y g
- (E) a x y g z

http://tiny.cc/cmps112-env-grp

Free vs bound variables

- An occurrence of `x` is **free** if it is not **bound**
- An occurrence of `x` is **bound** if it's inside
  - `e2` where `let x = e1 in e2`
  - `e` where `\x -> e`
- A closure environment has to save *all free variables* of a function definition!

```
let a = 20 in
let f =
  \x -> let y = x + 1 in
        let g = \z -> y + z in
        a + g x -- a is the only free variable!
  in ...
```


Evaluator

Let's modify our evaluator to handle functions!

```
data Value = VNum Int
           | VClos Env Id Expr -- env + formal + body

eval :: Env -> Expr -> Value
eval env (Num n)        = VNum n -- must wrap in VNum now!
eval env (Var x)        = lookup x env
eval env (Bin op e1 e2) = VNum (f v1 v2)
  where
    (VNum v1) = eval env e1
    (VNum v2) = eval env e2
    f = ... -- as before
eval env (Let x e1 e2) = eval env' e2
  where
    v = eval env e1
    env' = add x v env
eval env (Lam x body) = ??? -- construct a closure
eval env (App fun arg) = ??? -- eval fun, then arg, then apply
```


Evaluator

Evaluating functions:

- **Construct a closure**: save environment at function definition
- **Apply a closure**: restore saved environment, add formal, evaluate the body

```
eval :: Env -> Expr -> Value
...
eval env (Lam x body) = VClos env x body
eval env (App fun arg) = eval bodyEnv body
  where
    (VClos closEnv x body) = eval env fun -- eval function to closure
    vArg                   = eval env arg -- eval argument
    bodyEnv                = add x vArg closEnv
```

Quiz

With eval as defined above, what does this evaluate to? *

```
let f = \x -> x + y in
let y = 10 in
f 5
```

- (A) 15
- (B) 5
- (C) Error: unbound variable x
- (D) Error: unbound variable y
- (E) Error: unbound variable f

http://tiny.cc/cmps112-enveval-ind

Quiz

With eval as defined above, what does this evaluate to? *

```
let f = \x -> x + y in
let y = 10 in
f 5
```

- (A) 15
- (B) 5
- (C) Error: unbound variable x
- (D) Error: unbound variable y
- (E) Error: unbound variable f

http://tiny.cc/cmps112-enveval-grp

Evaluator

```
   eval []
     {let f = \x -> x + y in let y = 10 in f 5}
=> eval [f:<[], \x -> x + y>]
                             {let y = 10 in f 5}
=> eval [y:10, f:<[], \x -> x + y>]
                                          {f 5}
=> eval [y:10, f:<[], \x -> x + y>]
                           {<[], \x -> x + y> 5}
=> eval [x:5] -- env got replaced by closure env + formal!
                                      {x + y}   -- y is unbound!
```

Quiz

With eval as defined above, what does this evaluate to? *

```
let f = \n -> n * f (n - 1) in
f 5
```

- (A) 120
- (B) Evaluation does not terminate
- (C) Error: unbound variable f

http://tiny.cc/cmps112-enveval2-ind

Quiz

With eval as defined above, what does this evaluate to? *

```
let f = \n -> n * f (n - 1) in
f 5
```

- (A) 120
- (B) Evaluation does not terminate
- (C) Error: unbound variable f

http://tiny.cc/cmps112-enveval2-grp

Evaluator

```
   eval []
        {let f = \n -> n * f (n - 1) in f 5}
=> eval [f:<[], \n -> n * f (n - 1)>]
                                    {f 5}
=> eval [f:<[], \n -> n * f (n - 1)>]
              {<[], \n -> n * f (n - 1)> 5}
=> eval [n:5] -- env got replaced by closure env + formal!
                         {n * f (n - 1)} -- f is unbound!
```

Lesson learned: to support recursion, we need a different way of constructing the closure environment!

Nano1: Syntax

We need to define the syntax for *expressions* (*terms*) and *values* using a grammar:

```
e ::= n | x                 -- expressions
    | e1 + e2
    | let x = e1 in e2

v ::= n                     -- values
```

where $n \in \mathbb{N}$, $x \in Var$

Nano1: Operational Semantics

Operational semantics defines how to execute a program step by step

Let's define a *step relation* (*reduction relation*) `e => e'`

- "expression `e` makes a step (reduces in one step) to an expression `e'`

Nano1: Operational Semantics

We define the step relation *inductively* through a set of *rules*:

```
               e1 => e1'            -- premise
[Add-L]   --------------------
          e1 + e2 => e1' + e2       -- conclusion

              e2 => e2'
[Add-R]   --------------------
          n1 + e2 => n1 + e2'

[Add]     n1 + n2 => n         where n == n1 + n2

                        e1 => e1'
[Let-Def] ---------------------------------------
          let x = e1 in e2 => let x = e1' in e2

[Let]     let x = v in e2 => e2[x := v]
```

Nano1: Operational Semantics

Here `e[x := v]` is a value substitution:

```
x[x := v]                  = v
y[x := v]                  = y          -- assuming x /= y
n[x := v]                  = n
(e1 + e2)[x := v]          = e1[x := v] + e2[x := v]
(let x = e1 in e2)[x := v] = let x = e1[x := v] in e2
(let y = e1 in e2)[x := v] = let y = e1[x := v] in
e2[x := v]
```

Do not have to worry about capture, because `v` is a value (has no free variables!)

Nano1: Operational Semantics

A reduction is *valid* if we can build its **derivation** by "stacking" the rules:

```
    [Add] --------------------
                   1 + 2 => 3
[Add-L] -----------------------
        (1 + 2) + 5  =>  3 + 5
```

Do we have rules for all kinds of expressions?

Nano1: Operational Semantics

We define the step relation *inductively* through a set of *rules*:

```
              e1 => e1'         -- premise
[Add-L]   --------------------
          e1 + e2 => e1' + e2   -- conclusion

             e2 => e2'
[Add-R]   --------------------
          n1 + e2 => n1 + e2'

[Add]     n1 + n2 => n            where n == n1 + n2

                       e1 => e1'
[Let-Def] --------------------------------------
          let x = e1 in e2 => let x = e1' in e2

[Let]     let x = v in e2 => e2[x := v]
```

1. Normal forms

There are no reduction rules for:

- `n`
- `x`

Both of these expressions are *normal forms* (cannot be further reduced), however:

- `n` is a *value*
  - intuitively, corresponds to successful evaluation
- `x` is *not* a value
  - intuitively, corresponds to a run-time error!
  - we say the program `x` is **stuck**

2. Evaluation order

In `e1 + e2`, which side should we evaluate first?

In other words, which one of these reductions is valid (or both)?

```
1.(1 + 2) + (4 + 5) => 3 + (4 + 5)
2.(1 + 2) + (4 + 5) => (1 + 2) + 9
```

Reduction (1) is *valid* because we can build a **derivation** using the rules:

```
          [Add] ----------
                1 + 2 => 3
[Add-L] ----------------------------------
        (1 + 2) + (4 + 5)  =>  3 + (4 + 5)
```

Reduction (2) is *invalid* because we cannot build a derivation:

- there is *no rule* whose conclusion matches this reduction!

```
                     ???
[???] -----------------------------------
      (1 + 2) + (4 + 5)  =>  (1 + 2) + 9
```


Evaluation relation

Like in λ-calculus, we define the **multi-step reduction** relation `e =*> e'`:

`e =*> e'` iff there exists a sequence of expressions `e1, ..., en` such that

- `e = e1`
- `en = e'`
- `ei => e(i+1)` for each `i in [0..n)`

Example:

```
   (1 + 2) + (4 + 5)
=*> 3 + 9
```

because

```
   (1 + 2) + (4 + 5)
=> 3       + (4 + 5)
=> 3       + 9
```

Evaluation relation

Now we define the **evaluation relation** `e =~> e'`:

`e =~> e'` iff

- `e =*> e'`
- `e'` is in normal form

Example:

```
    (1 + 2) + (4 + 5)
=~> 12
```

because

```
   (1 + 2) + (4 + 5)
=> 3       + (4 + 5)
=> 3       + 9
=> 12
```

and `12` is a *value* (normal form)

Theorems about Nano1

Let's prove something about Nano1!

1. Every Nano1 program terminates
2. Closed Nano1 programs don't get stuck
3. *Corollary (1 + 2):* Every closed Nano1 program evaluates to a value

How do we prove theorems about languages?

By induction.

Mathematical induction in PL

1. Induction on natural numbers

To prove $\forall n.P(n)$ we need to prove:

- *Base case:* $P(0)$
- *Inductive case:* $P(n+1)$ assuming the *induction hypothesis* (IH): that $P(n)$ holds

Compare with inductive definition for natural numbers:

```
data Nat = Zero     -- base case
         | Succ Nat -- inductive case
```

No reason why this would only work for natural numbers…

In fact we can do induction on *any* inductively defined mathematical object (= any datatype)!

- lists
- trees
- programs (terms)
- etc

2. Induction on terms

```
e ::= n | x
    | e1 + e2
    | let x = e1 in e2
```

To prove $\forall e.P(e)$ we need to prove:

- *Base case 1:* `P(n)`
- *Base case 2:* `P(x)`
- *Inductive case 1:* `P(e1 + e2)` assuming the IH: that `P(e1)` and `P(e2)` hold
- *Inductive case 2:* `P(let x = e1 in e2)` assuming the IH: that `P(e1)` and `P(e2)`hold

3. Induction on derivations

Our reduction relation `=>` is also defined *inductively*!

- Axioms are bases cases
- Rules with premises are inductive cases

To prove $\forall e, e'.P(e \Rightarrow e')$ we need to prove:

- *Base cases:* `[Add]`, `[Let]`
- *Inductive cases:* `[Add-L]`, `[Add-R]`, `[Let-Def]` assuming the IH: that `P` holds of their premise

Theorem: Termination

Theorem I [Termination]: For any expression `e` there exists `e'` such that `e =~> e'`.

Proof idea: let's define the *size* of an expression such that

- size of each expression is positive
- each reduction step strictly decreases the size

Then the length of the execution sequence for `e` is *bounded* by the size of `e`!

```
size n                   = ???
size x                   = ???
size (e1 + e1)           = ???
size (let x = e1 in e2)  = ???
```


Theorem: Termination

Term size:

```
size n                   = 1
size x                   = 1
size (e1 + e1)           = size e1 + size e2
size (let x = e1 in e2)  = size e1 + size e2
```

Lemma 1: For any `e`, `size e > 0`.

Proof: By induction on the *term* `e`.

- *Base case 1:* `size n = 1 > 0`
- *Base case 2:* `size x = 1 > 0`
- *Inductive case 1:* `size (e1 + e2) = size e1 + size e2 > 0` because `size e1> 0` and `size e2 > 0` by IH.
- *Inductive case 2:* similar.

QED.

Theorem: Termination

Lemma 2: For any `e, e'` such that `e => e'`, `size e' < size e`.

Proof: By induction on the *derivation* of `e => e'`.

Base case `[Add]`.

- Given: the root of the derivation is

  `[Add]: n1 + n2 => n` where `n = n1 + n2`
- To prove: `size n < size (n1 + n2)`
- `size n = 1 < 2 = size (n1 + n2)`

Theorem: Termination

Lemma 2: For any `e, e'` such that `e => e'`, `size e' < size e`.

Inductive case `[Add-L]`.

- Given: the root of the derivation is `[Add-L]`:

```
    e1 => e1'
--------------------------
e1 + e2 => e1' + e2
```

- **To prove**: `size (e1' + e2) < size (e1 + e2)`
- **IH**: `size e1' < size e1`

```
  size (e1' + e2)
= -- def. size
  size e1' + size e2
< -- IH
  size e1 + size e2
= -- def. size
  size (e1 + e2)
```

Inductive case `[Add-R]`. Try at home

Theorem: Termination

Lemma 2: For any `e, e'` such that `e => e'`, `size e' < size e`.

Base case `[Let]`.

- Given: the root of the derivation is `[Let]`: `let x = v in e2 => e2[x := v]`
- To prove: `size (e2[x := v]) < size (let x = v in e2)`

```
  size (e2[x := v])
= -- auxiliary lemma!
  size e2
< -- IH
  size v + size e2
= -- def. size
  size (let x = v in e2)
QED.
```

Inductive case `[Let-Def]`. Try at home

Nano2: adding functions

Syntax

We need to extend the syntax of expressions and values:

```
e ::= n | x               -- expressions
    | e1 + e2
    | let x = e1 in e2
    | \x -> e         -- abstraction
    | e1 e2           -- application

v ::= n                   -- values
    | \x -> e         -- abstraction
```


Operational semantics

We need to extend our reduction relation with rules for abstraction and application:

```
           e1 => e1'
[App-L] ----------------
        e1 e2 => e1' e2

          e => e'
[App-R] ------------
        v e => v e'

[App]   (\x -> e) v => e[x := v]
```


Evaluation Order

```
   ((\x y -> x + y) 1) (1 + 2)
=> (\y -> 1 + y) (1 + 2)        -- [App-L], [App]
=> (\y -> 1 + y) 3              -- [App-R], [Add]
=> 1 + 3                        -- [App]
=> 4                            -- [Add]
```

Our rules define **call-by-value**:

1. Evaluate the function (to a lambda)
2. Evaluate the argument (to some value)
3. “Make the call”: make a substitution of formal to actual in the body of the lambda

The alternative is **call-by-name**:

- do not evaluate the argument before “making the call”
- can we modify the application rules for Nano2 to make it call-by-name?

Theorems about Nano2

Let's prove something about Nano2!

1. Every Nano2 program terminates (?)
2. Closed Nano2 programs don't get stuck (?)

Theorems about Nano2

1. Every Nano2 program terminates (?)

   What about `(\x -> x x) (\x -> x x)`?

2. Closed Nano2 programs don't get stuck (?)

   What about `1 2`?

Both theorems are now false!

To recover these properties, we need to add *types*:

1. Every *well-typed* Nano2 program terminates
2. *Well-typed* Nano2 programs don't get stuck

We'll do that next week!