cse116

Andrew Zhu

Dec 04, 2019

CONTENTS:

CHAPTER ONE

LAMBDA CALCULUS

Review: 9/26 - Lambda Calculus

Slides

occurrence
: an appearance of a variable in an expression (binding does not count)

1.1 Quizzes

tiny.cc/

cse116-lambda-ind -> A

cse116-scope-ind -> C

cse116-beta1-ind -> D

cse116-beta2-ind -> A

cse116-norm-ind -> C

cse116-church-ind -> A

cse116-add-ind -> A

cse116-mult-ind -> B

cse116-sum-ind -> NO

1.2 Reductions

alpha-reduction

`\x -> e =a> \y -> e[x := y] | where not (y in FV(e))`

beta-reduction

`(\x -> e1) e2 =b> e1[x := e2]`

"Replace all **free** occurrences of *x* in *e1* with *e2*."

```
x[x := e] = e
y[x := e] = y
(e1 e2)[x := e] = (e1[x := e]) (e2[x := e])
(\x -> e1)[x := e] = \x -> e1                    -- since x in e1 is bound
```

(continues on next page)

(continued from previous page)

```
(\y -> e1)[x := e]
  | not (y in FV(e)) = \y -> e1[x := e]
  | otherwise undefined
```

1.3 Normal Forms

A **redex** is a lambda-term of the form `(\x -> e1) e2` (i.e. can be beta-reduced).

A lambda-term is in **normal form** if it contains no redexes (i.e. cannot be beta-reduced).

1.4 Semantics: Evaluation

A lambda-term *e* evaluates to *e'* if: 1. There is a sequence of stops `e =?> e_1 =?> ... =?> e'`

1.4.1 Examples

```
(\x -> x) apple
    =b> apple

(\f -> f (\x -> x)) (\x -> x)
    =b> (\x -> x) (\x -> x)
    =b> \x -> x

(\x -> x x) (\x -> x)
    =b> (\x -> x) (\x -> x)
    =b> \x -> x
```

1.4.2 Elsa Shortcuts

1.4.2.1 Named lambda-terms

`let ID = \x -> x`

To substitute a name with its defn, use a =d> step

```
ID apple
    =d> \x -> x apple
    =b> apple
```

1.4.2.2 Evaluation

`e1 =*> e2` - e1 reduces to e2 in 0 or more steps, where each step is in =a>, =b>, =d>

`e1 =~> e2` - e1 evaluates to e2 (i.e. final output)

1.4.3 Non-Terminating Evaluation

```
(\x -> x x) (\x -> x x)
    =b> (\x -> x x) (\x -> x x)
```

Programs can loop and never reduce to normal form!

This is called the omega-term.

What if we pass omega to another function?

```
let OMEGA = (\x -> x x) (\x -> x x)
(\x -> \y -> y) OMEGA
```

1.5 Lambda Calculus: Booleans

How do we encode T/F as a func?

With booleans, we make a binary choice (e.g. `if b then e1 else e2`)

We need to define three functions:

```
let TRUE = \x y -> x
let FALSE = \x y -> y
let ITE = \b x y -> b x y
```

such that

```
ITE TRUE apple banana =~> apple
ITE FALSE apple banana =~> banana
```

```
eval ite_true:
    ITE TRUE e1 e2
    =d> (\b x y -> b x y)      TRUE e1 e2
    =b>   (\x y -> TRUE x y)   e1 e2
    =b>     (\y -> TRUE e1 y) e2
    =b>            TRUE e1 e2
    =d> (\x y -> x) e1 e2
    =b>   (\y -> e1) e2
    =b> e1

eval ite_false:
    ITE FALSE e1 e2
    =d> (\b x y -> b x y)       FALSE e1 e2
    =b>   (\x y -> FALSE x y)   e1 e2
    =b>     (\y -> FALSE e1 y) e2
    =b>            FALSE e1 e2
    =d> (\x y -> y) e1 e2
    =b>   (\y -> y) e2
    =b> e2
```

Now we can define other boolean operators:

```
let NOT = \b     -> ITE b FALSE TRUE
let AND = \b1 b2 -> ITE b1 b2 FALSE
let OR  = \b1 b2 -> ITE b1 TRUE b2
```

(`ITE` is redundant, so it can be removed from these defns)

1.6 Lambda Calculus: Records

- Start with records w/ 2 fields (pairs)
- **What do we want to do?**
  - Pack two items into a pair
  - Get first
  - Get second

1.6.1 API

```
let PAIR = \x y -> (\b -> ITE b x y)
    -- a function that returns a function
        -- that takes a boolean asking which item you want
let FST  = \p -> p TRUE
let SND  = \p -> p FALSE
```

such that

```
FST (PAIR apple banana) =~> apple
SND (PAIR apple banana) =~> banana
```

1.6.2 Triples

```
let TRIPLE = \x y z -> PAIR x (PAIR y z)
let FST3   = \t -> FST t
let SND3   = \t -> FST (SND t)
let TRD3   = \t -> SND (SND t)
```

1.7 Lambda Calculus: Numbers

- What about natural numbers [0..]?
- Counters, arithmetic, comparisons
- +, -, *, ==, <=, etc

We need to define:

- a family of numerals `ZERO, ONE, TWO`, etc
- arithmetic functions `INC, DEC, ADD, SUB, MULT`
- comparisons `IS_ZERO, EQ`

1.7.1 Implementation

Church numerals: A number N is encoded as a combinator that calls a function on an argument N times

```
let ZERO  = \f x -> x
let ONE   = \f x -> f x
let TWO   = \f x -> f (f x)
let THREE = \f x -> f (f (f x))
...etc
```

1.7.1.1 Increment

```
-- call `f` on `x` one more time than `n` does
let INC = \n -> (\f x -> f (n f x))

-- ex
INC ZERO
    =d> (\n f x -> f (n f x)) ZERO
    =b> \f x -> f (ZERO f x)
    =*> \f x -> f x
    =d> ONE
```

1.7.1.2 Add

```
let ADD = \n m -> n INC m
-- n is a function that takes a function and number
-- i.e. apply INC n times to m

-- ex
eval add_one_zero:
    ADD ONE ZERO
        =d> (\n m -> n INC m) ONE ZERO
        =b> (\m -> ONE INC m) ZERO
        =b> ONE INC ZERO
        =d> (\f x -> f x) INC ZERO
        =b> INC ZERO
        =*> ONE

eval add_two_one:
    ADD TWO ONE
        =d> (\n m -> n INC m) TWO ONE
        =b> (\m -> TWO INC m) ONE
        =b> TWO INC ONE
        =d> (\f x -> f (f x)) INC ONE
        =b> INC (INC ONE)
        =*> THREE
```

1.7.1.3 Mult

```
let MULT = \n m -> n (ADD m) ZERO
-- ADD m returns a function
-- so we call ADD m on ZERO n times
```

(continues on next page)

(continued from previous page)

```
-- similar to python partials

-- ex
eval two_times_one:
    MULT TWO ONE
        =d> (\n m -> n (ADD m) ZERO) TWO ONE
        =b> (\m -> TWO (ADD m) ZERO) ONE
        =b> TWO (ADD ONE) ZERO
        =~> ADD ONE (ADD ONE ZERO)
        =~> TWO
```

1.8 Lambda Calculus: Recursion

Ex. I want to write a number that sums up natural numbers to n.

- `\n -> ... -- = 1 + 2 + ... + n`

Step 1: Pass in the function to call recursively

```
let STEP =
    \rec ->
        \n -> ITE (ISZ n)
            ZERO
            (ADD n (rec (DEC n)))
```

Step 2: Do something to `STEP` so that the function passed as `rec` becomes:

`\n -> ITE (ISZ n) ZERO (ADD n (rec (DEC n)))`

Note: Wanted: a combinator `FIX` s.t. `FIX STEP` calls `STEP` with itself as the first argument

```
FIX STEP
    =*> STEP (FIX STEP)
```

Note: It's important that `STEP` has some base case in it, or else you end up with `STEP (STEP (STEP (STEP...)))`

---

then, `let SUM = FIX STEP`, so `SUM =*> STEP SUM`

```
eval sum_one:
    SUM ONE
        =*> STEP SUM ONE
        =d> (\rec n -> ITE (ISZ n) ZERO (ADD n (rec (DEC n)))) SUM ONE
        =b> (\n -> ITE (ISZ n) ZERO (ADD n (SUM (DEC n)))) ONE
        =b> ITE (ISZ ONE) ZERO (ADD ONE (SUM (DEC ONE)))
        =*> ITE FALSE ZERO (ADD ONE (STEP SUM ZERO))
        =*> ADD ONE (SUM ZERO)
        =*> ADD ONE (STEP SUM ZERO)
        =d> ADD ONE ((\rec n -> ITE (ISZ n) ZERO (ADD n (rec (DEC n)))) SUM ZERO)
        =b> ADD ONE ((\n -> ITE (ISZ n) ZERO (ADD n (SUM (DEC n)))) ZERO)
        =b> ADD ONE (ITE (ISZ ZERO) ZERO (ADD ZERO (SUM (DEC ZERO))))
        =*> ADD ONE (ITE TRUE ZERO (ADD ZERO (SUM (DEC ZERO))))
```

(continues on next page)

(continued from previous page)

```
        =*> ADD ONE ZERO
        =~> ONE
```

So how do we define `FIX`?

- **Let's look back at omega:**
  - `(\x -> x x) (\x -> x x) =b> (\x -> x x) (\x -> x x)`
- We need something similar but with control
- Thus, the Y combinator (or fixpoint)

```
let FIX = \stp -> (\x -> stp (x x)) (\x -> stp (x x))

eval fix_step:
    FIX STEP
    =d> (\stp -> (\x -> stp (x x)) (\x -> stp (x x))) STEP
    =b> (\x -> STEP (x x)) (\x -> STEP (x x))
    =b> STEP ((\x -> STEP (x x)) (\x -> STEP (x x)))
    =d> STEP (FIX STEP)
```

Note: Example: `MULT` using recursion

```
-- if we can use recursion by name:
let MULT x y =
    ITE (ISZ y)
        ZERO
        ADD x (MULT x (DECR y))

-- replace the self ref with a passed func
let MULT1 f x y =
    ITE (ISZ y)
        ZERO
        ADD x (f x (DECR y))

-- and use fixpt
let MULT = FIX MULT1

-- therefore, generally
let FUNC0 = \f n -> ... f (DECR n)
let FUNC = FIX FUNC0
```

CHAPTER TWO

HASKELL

Slides

2.1 Quizzes

cse116-pair-ind -> D

cse116-tpair-ind -> D

cse116-pattern-ind -> D

2.2 What is Haskell?

Haskell is a typed, lazy, purely functional language, with:

- types
- builtins (bools, numbers, chars)
- tuples
- lists
- recursion

Haskell v. lambda-calc:

- A program is an expression, not a sequence of statements
- it evaluates to a value, does not perform actions
- **functions are first-class values**
  - can be passed as args
  - can be returned from a func
  - can be partially applied
- **but there are things that aren't funcs**
  - variable assignments/literals
  - top level bindings
- you can also define funcs using equations

```
pair x y b = if b then x else y -- \x y b -> ITE b x y
```

- and patterns:

```
pair x y True  = x
pair x y False = y
```

- a pattern is a variable (matches any value), or a value (matches that value)
- the above pattern is equivalent to:

```
pair x y True  = x
pair x y b     = y

pair x y True  = x
pair x y _     = y -- wildcard: don't create binding
```

2.3 Guards

- an expression can have multiple guards (bool exp)

```
cmpSquare x y | x > y*y  = "bigger"
              | x == y*y = "equal"
              | x < y*y  = "smaller"

-- equals to
cmpSquare x y | x > y*y   = "bigger"
              | x == y*y  = "equal"
              | otherwise = "smaller"
```

2.4 Recursion

Is built in!

```
sum n = if n == 0
            then 0
            else n + sum (n - 1)

-- or
sum 0 = 0
sum n = n + sum (n - 1)
```

2.5 Variable Scope

- Top level vars have global scope

```
-- vars defined out of order
message = if foo
            then "bar"
            else "baz"
```

(continues on next page)

(continued from previous page)

```
foo = True

-- mutual recursion
f 0 = True
f n = g (n - 1)

g 0 = False
g n = f (n - 1)

-- this is not allowed: immutable vars, can only be defined once per scope
foo = True
foo = False
```

2.5.1 Local Variables

You can introduce a new local scope using a let-expression

```
sum 0 = 0
sum n = let n' = n - 1   -- n' is only in scope in the in block
        in n + sum n'

-- multiple lets
sum 0 = 0
sum n = let
            n' = n - 1
            sum' = sum n'
        in n + sum'
```

If you need a var whose scope is an eqn, use `where`

```
cmpSquare x y | x > z  = "bigger"
              | x == z = "equal"
              | x < z  = "smaller"
    where z = y*y
```

2.6 Types

Lambda-calculus is untyped: for example, `let FNORD = ONE ZERO`.

In Haskell, every expression either has a type or is **ill-typed** and rejected statically (at compile-time)

2.6.1 Type Annotations

You can annotate bindings with types using `::`

```
foo :: Bool
foo = True

message :: String
message = if foo
            then "bar"
```

(continues on next page)

(continued from previous page)

```
           else "baz"

-- word-sized integer
rating :: Int
rating = if foo then 10 else 0

-- arbitrary precision int
something :: Integer
something = factorial 100
```

Functions have arrow types

```
> :t (\x -> if x then 'a' else 'b')
(\x -> if x then 'a' else 'b') :: Bool -> Char

-- annotate function bindings!
sum :: Int -> Int
sum 0 = 0
sum n = n + sum (n - 1)

-- multiple args
pair :: String -> (String -> (Bool -> String))
pair x y b = if b then x else y

-- same as
pair :: String -> String -> Bool -> String
pair x y b = if b then x else y
```

2.6.2 Lists

A list is:

```
-- an empty list
[] -- "nil"

-- a head element attached to a tail list
x:xs -- "x cons xs"

-- examples
[] -- a list with 0 elements

1:[] -- [1]

(:) 1 [] -- for any infix op, (op) is a regular function

1:(2:(3:(4:[]))) -- [1, 2, 3, 4]

1:2:3:4:[] -- same as above

[1,2,3,4] -- guess what this does
```

`[]` and `(:)` are the list constructors

- `True` and `False` are `Bool` constructors
- `0, 1, 2` are… complicated, but basically `Int` constructors

- they take 0 args, so we call them values

A list has type `[A]` when each of its elements has type `A`

```
foo :: [Int]
foo = [1,2,3]

bar :: [Char]                   -- = String
bar = ['h', 'e', 'l', 'l', 'o'] -- = "hello"

generic :: [t]
generic = []
```

2.6.2.1 Functions on List

```
-- range
upto :: Int -> Int -> [Int]
upto n m
    | n > m      = []
    | otherwise = n : (upto (n + 1) m)

-- syntactic sugar:
[1..7]    -- = [1,2,3,4,5,6,7]
[1,3..7] -- = [1,3,5,7]

-- length
length :: [Int] -> Int
length []      = 0
length (_:xs) = 1 + length xs  -- note: a pattern can be applied to other patterns
```

Pattern matching attempts to match values against patterns and, if desired, bind variables to successful values

2.6.2.2 List Comprehensions

```
[toUpper c | c <- s]
-- [toUpper(char) for c in s] in Python

[(i, j) | i <- [1..3],
          j <- [1..i]) -- multiple generators
-- [(i, j) for i in range(1, 4) for j in range(1, i+1)]

[(i, j) | i <- [1..3],
          j <- [1..i],
          i + j == 5) -- multiple generators with condition
-- [(i, j) for i in range(1, 4) for j in range(1, i+1) if i + j == 5]
```

2.6.3 Pairs

```
myPair :: (String, Int)
myPair = ("apple", 3)
```

`(,)` is the pair constructor

```
-- field access
fruit = fst myPair
num   = snd myPair

-- field access using patterns
isEmpty (x, y) = y == 0

-- same as
isEmpty         = \(x, y) -> y == 0
isEmpty p       = let (x, y) = p in y == 0
```

What about:

```
f :: String -> [(String, Int)] -> Int
f _ [] = 0
f x ((k,v) : ps)
    | x == k     = v
    | otherwise = f x ps

-- in Python: f = ((k,v) : ps).get(x, 0)
-- key-value pair lookups
```

2.6.4 Tuples

Go ahead and make n-tuples, they work pretty much as you expect

```
triple :: (Bool, Int, [Int])
triple = (True, 1, [1,2,3])

-- also
myUnit :: ()
myUnit = ()
```

CHAPTER

THREE

DATATYPES AND RECURSION

Slides

Quizzes

cse116-para-ind -> C cse116-adt-ind -> D cse116-case-ind -> B cse116-case2-ind -> D cse116-rectype-ind -> E cse116-tree-ind -> C cse116-leaves-ind -> D cse116-tail-ind -> NO

3.1 Representing complex data

- base/primitive types: int, float, bool, etc
- ways to build up types: functions, tuples, lists

Algebraic Data Types: a technique to build data types from these

---

Note: Tuples can do the job, but there are two problems:

- verbose and unreadable
- no type checking (unsafe)

```
type Date = (Int, Int, Int)
type Time = (Int, Int, Int)

deadDate :: Date
deadDate = (2, 4, 2019)

deadTime :: Time
deadTime = (11, 59, 59)

-- example: extend
extension :: Date -> Date
extension = ...

-- however, you can do
extension deadTime -- which should error!
```

Solution: construct *datatypes*

```
data Date = Date Int Int Int
data Time = Time Int Int Int
-- constructor ^   ^ param types
```

(continues on next page)

(continued from previous page)

```
deadDate :: Date
deadDate = Date 2 4 2019

deadTime :: Time
deadTime = Time 11 59 59
```

3.2 Building Data Types

1. Product types (each-of): a value of `T` contains a value of `T1` and a value of `T2`
2. Sum types (one-of): A value of `T` contains a value of `T1` *or* a value of `T2`
3. Recursive types: A value of `T` contains subvalues of type `T`

3.2.1 Product Types

You can name the constructor params:

```
data Date = Date {
    month :: Int,
    day :: Int,
    year :: Int
}

deadDate = Date 2 4 2019
deadMonth = month deadDate
-- field name is func that accesses date
```

3.2.2 Sum Types

e.g. a type for `Paragraph` that is one of the three options

```
data Paragraph =
      Text String
    | Heading Int String
    | List Bool [String]
```

3.2.3 Recursive Types

See recursive-types

3.3 Constructing Datatypes

```
data T =
      C1 T11 .. T1k
```

C2 T21 .. T2l
..
Cn Tn1 .. Tnm

`T` is the **datatype**

`C1 .. Cn` are the **constructors**

A **value** of type `T` is

- either `C1 v1 .. vk` with `vi :: T1i`
- or `C2 v1 .. vl` with `vi :: T2i`
- or ...
- or `Cn v1 .. vm` with `vi :: Tni`

3.4 Writing Functions

e.g. how to write a function to convert nanoMD to HTML?

3.4.1 Pattern Matching

match on the constructor

```
html :: Paragraph -> String
html (Text str) = ...
html (Heading lvl str) = ...
html (List ord items) = ...
```

But, there are dangers:

```
-- example: missing a type
html :: Paragraph -> String
html (Text str) = ...
html (List ord items) = ...

html (Heading 1 "Introduction") -- runtime error!
```

You can also pattern match inside the program:

```
html :: Paragraph -> String
html p =
    case p of
        Text str -> ...
        Heading lvl str -> ...
        List ord items -> ...
```

3.4.2 Case

```
case e of
    pattern1 -> e1
    pattern2 -> e2
    ...
    patternN -> eN
```

has type `T` if:

- each `e1..eN` has type `T`
- `e` has some type `D`
- each `pattern1..patternN` is a valid pattern for `D`

3.5 Recursive Types

Let's define natural numbers.

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

Zero      -- 0
Succ Zero -- 1
```

A `Nat` value is a box named `Zero` or a box labeled `Succ` with another `Nat` in it

3.5.1 Using as Parameter

```
toInt :: Nat -> Int
toInt Zero     = 0            -- base case
toInt (Succ n) = 1 + toInt n -- inductive case
```

3.5.2 Using as Result

```
fromInt :: Int -> Nat
fromInt n
    | n <= 0    = Zero
    | otherwise = Succ (fromInt (n - 1))

-- and operations
add :: Nat -> Nat -> Nat
add Zero     m = m
add (Succ n) m = Succ (add n m)

sub :: Nat -> Nat -> Nat
sub n        Zero     = n
sub Zero     _        = Zero
sub (Succ n) (Succ m) = sub n m
```

3.5.3 Lists

Lists aren't built in!

```
data List = Nil
    | Cons Int List

[1, 2, 3] == Cons 1 (Cons 2 (Cons 3 Nil))
```

Ex. appending two lists

```
append :: List -> List -> List
append [] ys     = ys
append (x:xs) ys = x:(append xs ys)

append2 :: List -> List -> List
append2 xs []     = xs
append2 xs (y:ys) = append xs:y ys
```

3.5.4 Trees

Think of lists as unary trees with elements stored in the nodes. What about binary trees?

```
data Tree = Leaf | Node Int Tree Tree  -- leaves don't store data!

t1234 = Node 1
            (Node 2 (Node 3 Leaf Leaf) Leaf)
            (Node 4 Leaf Leaf)

1 - 2 - 3 - ()
  |   |   \ ()
  |   \ ()
  \ 4 - ()
      \ ()
```

3.5.4.1 Functions on Trees

```
depth :: Tree -> Int
depth Leaf = 0
depth (Node _ l r) = 1 + max (depth l) (depth r)
```

3.5.4.2 Ex: Calculator

Let's implement an arithmetic calculator to eval things like 4.0 + 2.0, 3 - 9, (4.0 + 2.9) * (1.0 + 2.2)

```
data Expr = Val Float
          | Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr

-- evaluate!
eval :: Expr -> Float
eval (Num f)     = f
eval (Add e1 e2) = eval e1 + eval e2
eval (Sub e1 e2) = eval e1 - eval e2
eval (Mul e1 e2) = eval e1 * eval e2
```

3.6 Tail Recursion

Whatever the recursive call returns will be what the expression returns. No computations are allowed on recursively returned values.

```
-- tail recursive factorial!
facTR :: Int -> Int
facTR n = loop 1 n
    where
        loop :: Int -> Int -> Int
        loop acc n
            | n <= 1     = acc
            | otherwise = loop (acc * n) (n - 1)

--      <facTR 4>
--     <<loop 1 4>>
--    <<<loop 4 3>>>
--   <<<<loop 12 2>>>>
--  <<<<<loop 24 1>>>>>
-- <<<<<<24>>>>>>
```

CHAPTER

FOUR

HIGHER-ORDER FUNCTIONS

4.1 Intro

Slides

Quizzes

cse116-map-ind -> D

cse116-quiz-ind -> D

cse116-foldeval-ind -> B

cse116-foldtype-ind -> D

cse116-foldl2-ind ->

In this lecture: code reuse with higher-order functions (HOFs)

e.g.: map, filter, fold

4.1.1 Recursion

Gets pretty old pretty quickly!

Ex. a function that finds all even nums in a list

```
evens :: [Int] -> [Int]
evens []                       = []
evens (x:xs) | x `mod` 2 == 0 = x:(evens xs)
             | otherwise       = evens xs
```

or a function that filters 4 letter words

```
fourChars :: [Int] -> [Int]
fourChars []      = []
fourChars (x:xs) | (length x) == 4 = x:(fourChars xs)
                 | otherwise       = fourChars xs
```

4.1.2 HOFs

HOFs are a general pattern expressed as a HOF that takes customizable args, applied multiple times

4.2 Filter

```
filter :: (a -> Bool) -> [a] -> [a]  -- polymorphic type!
filter f [] = []
filter f (x:xs)
    | f x       = x:(filter f xs)
    | otherwise = filter f xs

-- now we can:
evens = filter isEven
    where isEven x = x `mod` 2 == 0

fourChars = filter isFour
    where isFour x = length x == 4
```

4.3 Map

Ex: we want to do some op on every elem

```
-- boring!
shout []     = []
shout (x:xs) = toUpper x : shout xs

square []     = []
square (x:xs) = x * x : square xs
```

Let's do this!

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs

-- so
shout = map (\x -> toUpper x)
square = map (\x -> x*x)
```

4.4 Fold

Ex: length/sum of a list

How about joining a list of strings?

```
cat :: [String] -> String
cat []     = ""
cat (x:xs) = x ++ cat xs
```

4.4.1 Fold-Right

This is fold-right!

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f b []     = b
foldr f b (x:xs) = f x (foldr f b xs)

-- so:
sum = foldr (+) 0
cat = foldr (++) ""
len = foldr (\x n -> 1 + n) 0
```

It's called this because it accumulates from the right (expansion is right associative)

4.4.2 Fold-Left

What about tail recursive versions?

```
-- tail recursive cat!
catTR :: [String] -> String
catTR xs = helper "" xs
    where
        helper acc []     = acc
        helper acc (x:xs) = helper (acc ++ x) xs
```

so:

```
foldl :: (a -> b -> b) -> b -> [a] -> b
foldl f b xs = helper b xs
    where
        helper acc []     = acc
        helper acc (x:xs) = helper (f acc x) xs

-- so, syntax is the same as foldr:
sumTR = foldl (+) 0
catTR = foldl (++) ""
```

4.5 Flip

Useful HOF:

```
-- instead of writing:
foldl (\xs x -> x : xs) [] [1, 2, 3]

-- write:
foldl (flip (:)) [] [1, 2, 3]

flip :: (a -> b -> c) -> (b -> a -> c)
```

4.6 Compose

```
map (\x -> f (g x)) ys
-- ==
```

(continues on next page)

(continued from previous page)

```
map (f . g) ys

(.) :: (b -> c) -> (a -> b) -> a -> c
```

CHAPTER

FIVE

ENVIRONMENTS & CLOSURES

5.1 Intro

Slides

Quizzes

cse116-vars-ind -> E

cse116-free-ind -> B

cse116-cscope-ind -> B

cse116-env-ind -> A

cse116-enveval-ind -> D

cse116-enveval2-ind -> C

5.2 The Nano Language

Features of Nano:

5.2.1 1. Arithmetic expressions

5.2.1.1 Evaluator 1

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

-- haskell representation:
data Binop = Add | Sub | Mul
data Expr = Num Int
            | Bin Binop Expr Expr

-- evaluator:
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
```

5.2.2 2. Variables and let-bindings

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

-- haskell representation:
type Id = String

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

thus, expressions must be evaluated in *Environments*

5.2.2.1 Evaluator 2

Previous implementation: *Evaluator 1*

```
type Value = Int
data Env = ...

-- add new id/value to env
add :: Id -> Value -> Env -> Env

-- lookup id in env
lookup :: Id -> Env -> Value

-- evaluator:
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
```

5.2.2.2 Runtime Errors

Lookups can fail when a var is not bound!

How do we ensure that it doesn't raise a runtime error?

In `eval env e`, *env* must contain bindings for all free vars of *e*. Evaluation only succeeds when all expressions are closed.

5.2.3 3. Functions

Let's add lambda abstractions and function application!

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

-- haskell representation:
data Expr = Num Int                  -- number
            | Var Id                 -- variable
            | Bin Binop Expr Expr    -- binary op
            | Let Id Expr Expr       -- let expr
            | Lam Id Expr            -- abstraction
            | App Expr Expr          -- application
```

Note: Now, let's try to evaluate something...

```
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}
```

How do we represent lambdas as a value? Let's try `data Value = VNum Int | VLam Id Expr` and evaluate...

```
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}
=> eval [cTimes:(\x -> c * x), c:42] {(\x -> c * x) 2}
=> eval [x:2, cTimes:(\x -> c * x), c:42] {x * c}
=> 42 * 2
=> 84
```

But what if *c* is redefined before *cTimes* is used?

The problem that this brings up is **static v. dynamic** scoping; static scoping = most recent binding in text, whereas dynamic = most recent binding in execution

---

How do we implement lexical scoping? See *Closures*

Now let's update our evaluator! Previous implementation: *Evaluator 2*

5.2.3.1 Evaluator 3

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

eval :: Env -> Expr -> Value
eval env (Num n)           = VNum n   -- we 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
```

(continues on next page)

(continued from previous page)

```
        (VNum 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
-- new!
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
```

But note: this evaluator doesn't cover recursion!

5.2.4 4. Recursion

We have to do this in homework, yay! See hw4.

5.3 Environments

an environment maps all free vars to values

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

x * y
=[x:17]=> Error: unbound var y

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

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

- evaluate `e2` in an extended env `env + [x:v]`
- where `v = eval e1`

5.4 Closures

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

a closure environment must save all free variables of a function defn!

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

-- our syntax:
```

(continues on next page)

(continued from previous page)

```
-- binding:<env, lambda>

-- now, eval:
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:<[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
```

CHAPTER SIX

THEOREMS ABOUT PROGRAMS

6.1 Intro

Slides

Quizzes

cse116-reduce-ind -> A

cse116-induct-ind -> B

cse116-reduce2-ind -> E

cse116-nano2-ind -> D

```
[Add]      --------------------------------------------------------
                                1 + 2 => 3
[Let-Def]  --------------------------------------------------------
           (let x = 1 + 2 in 4 + 5 + x) => (let x = 3 in 4 + 5 + x)
```

6.2 Formalizing Nano

We want to be able to guarantee properties about programs, such as:

- evaluation is deterministic
- all programs terminate
- certain programs never fail at runtime
- etc.

To prove theorems about programs we first need to define formally

- their syntax (what programs look like)
- their semantics (what it means to run a program)

Let's start with Nano1 (Nano w/o functions) and prove some stuff!

6.2.1 Nano1: Syntax

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

v ::= n                                 --  values
```

where n , x Var

6.2.2 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'

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]
```

and we can define `e[x := v]` as:

```
x[x := v]                   = v
y[x := v]                   = 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 x = e1[x := v] in e2[x := v]
```

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
```

Note: we don't have reduction rules for *n* or *x*, since both these expressions cannot be further reduced (normal).

However, *x* is not a value, and if the final result is that, it's a runtime error (**stuck**)

6.2.3 Evaluation Order

Out of these expressions, only the first is valid:

- `(1 + 2) + (3 + 4) => 3 + (3 + 4)`
- `(1 + 2) + (3 + 4) => (1 + 2) + 7`

since expression 1 has a derivation, but expr 2 does not:

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

-- but:
  [???] ---------------------------------
        (1 + 2) + (3 + 4) => (1 + 2) + 7
```

6.2.4 Evaluation Relation

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

`e =*> e'` iff there exists a sequence of expressions `e1..en` s.t. `` `e1 = e``, `en = e'`, `ei => e(i+1)`

Similarly, we can define **evaluation relations** `e =~> e'`.

6.2.5 Nano1 Thms

Let's prove:

- every Nano1 program terminates
- Closed Nano1 programs don't get stuck
- (corollary 1+2): closed nano programs evaluate to a value

using induction!

6.2.5.1 Induction on terms

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

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

- BS 1: P(n)
- BS 2: P(x)
- IS 1: P(e1 + e2) assuming P(e1) and P(e2)
- IS 2: P(let x = e1 in e2) assuming P(e1) and P(e2)

6.2.5.2 Induction on derivations

The relation => is also defined inductively:

- axioms are base cases (`[Add]`, `[Let]`)
- rules with premises are inductive cases (`[Add-L]`, `[Add-R]`, `[Let-Def]`)

6.2.6 Thm: Termination

Thm 1: For any expression `e`, there exists `e'` s.t. `e =~> e'`.

Let's define the size of an expression s.t.:

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

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

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

- BS 1: `size n = 1 > 0`.
- BS 2: `size x = 1 > 0`.
- IS 1: `size (e1 + e2) = size e1 + size e2 > 0` because `size e1 > 0` and `size e2 > 0` by IH.
- IS 2: similar.

Lemma 2: For any `e, e'` s.t. `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)`
- `1 < 2`.

Inductive case: [Add-L]

- Given: the root of the derivation is `[Add-L]`: (defn [Add-L].)
- To prove: `size (e1' + e2) < size (e1 + e2)`
- IH: `size e1' < size e1`
- `size e1' + size e2 < size e1 + size e2` by addition
- `size (e1' + e2) < size (e1 + e2)` by defn of size. QED.

Base case: [Let]

- Given: root of the derivation is `[Let]: let x = v in e2 => e2[x := v]`
- Prove: `size (e2[x := v]) < size (let x = v in e2)`
- `size (e2[x := v]) = size e2` by aux lemma
- `size (let x = v in e2) = size v + size e2` by defn
- `size e2 < size v + size e2` by lemma 1
- therefore, `size (e2[x := v]) < size (let x = v in e2)`

6.3 Nano2: Adding functions

Let's extend the syntax:

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

v ::= n | (\x -> e)
```

6.3.1 Operational Semantics

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

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

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

example:

```
((\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 implement call-by-value:

- evaluate the function (to a lambda)
- evaluate the arg (to some value)
- make the call: make a sub of formal to actual in body

the alternative is call-by-name:

- do not evaluate the argument before making the call
- let's modify the rules to make it call by name!

modified call-by-name:

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

[App] (\x -> e1) e2 => e1[x := e2]
```

6.3.2 Thms about Nano2

- not every program will terminate! think of the omega term

- programs can get stuck! what about `1 2`?

CHAPTER

SEVEN

POLYMORPHISM & TYPE INFERENCE

7.1 Intro

Slides

Quizzes

cse116-nanotype-ind -> D1

cse116-typed-ind -> B

cse116-subst-ind -> B

cse116-unify-ind -> C, D, E

cse116-infer-ind -> E

7.2 Type System

A type system defines what types an expression can have

To define a type system, we need to define:

- the syntax of types: what do types look like?
- the static semantics of our language (i.e. the typing rules): assign types to expressions

7.2.1 Syntax of Types

```
T ::= Int         -- integers
    | T1 -> T2    -- function types
```

Now, we define a typing relation `e :: T` ("e has type T"), inductively thru typing rules:

```
[T-Num] n :: Int

        e1 :: Int     e2 :: Int   -- premises
[T-Add] -----------------------
            e1 + e2 :: Int        -- conclusions

[T-Var] x :: ???
```

7.2.2 Type Environment

An expression has a type in a given type environment (or context), which maps all its free variables to their types:

```
G = x1:T1, x2:T2, ..., xn:Tn

-- now, our typing relation should include G:
G |- e :: T  -- e has type T in G
```

7.2.3 Typing Rules

An expression e has type T if we can derive `G |- e :: T` using these rules

An expression e is well-typed in G if we can derive `G |- e :: T` for some type T

```
-- typing rules using G
[T-Num] G |- n :: Int

        G |- e1 :: Int    G |- e2 :: Int
[T-Add] --------------------------------
            G |- e1 + e2 :: Int

[T-Var] G |- x :: T               if x:T in G

           G,x:T1 |- e :: T2
[T-Abs] ------------------------
        G |- \x -> e :: T1 -> T2

        G |- e1 :: T1 -> T2     G |- e2 :: T1
[T-App] ------------------------------------   -- modus ponens!
                G |- e1 e2 :: T2

        G |- e1 :: T1     G,x:T1 |- e2 :: T2
[T-Let] -----------------------------------
           G |- let x = e1 in e2 :: T2
```

Note: examples:

```
-- 1
[] |- (\x -> x) 2 :: Int

[T-Var]  -------------------
         [x:Int] |- x :: Int
[T-Abs]  -------------------             --------------  [T-Num]
         [] |- \x -> x :: Int -> Int      [] |- 2 :: Int
[T-App]  -----------------------------------------------
         [] |- (\x -> x) 2 :: Int

-- 2
[] |- let x = 1 in x + 2 :: Int

[T-Var] -----------------    -----------------[T-Num]
        x:Int |- x :: Int    x:Int |- 2 :: Int
[T-Num] --------------    ------------------------------------[T-Add]
        [] |- 1 :: Int    x:Int |- x + 2 :: Int
```

(continues on next page)

(continued from previous page)

```
[T-Let] -----------------------------------
        [] |- let x = 1 in x + 2 :: Int
```

`[] |- (\x -> x x) :: T` is underivable, because T has to be equal to `T -> T`

According to these rules, an expression can have zero, one, or many types.

e.g. `1 2` has no types, `1` has 1 type, `\x -> x` has many types.

One problem with this system: there's no generics.

7.3 Polymorphic Types

We can formalize a type `a -> a` as a polymorphic type: `forall a . a -> a`

- where `a` is a bound type variable
- also called a type scheme
- haskell has polymorphic types, but forall isn't usually required

We can instantiate this scheme into different types by replacing `a` in the body with some type, e.g. instantiating with `Int` yields `Int -> Int`.

Note: Similar to lambda expression at type level

With polymorphic types, we can derive `e :: Int -> Int` where `e` is

```
let id = \x -> x in
    let y = id 5 in
        id (\z -> z + y)
```

Inference works as follows:

1. When we have to pick a type T for x, we pick a fresh type variable a
2. So the type of `\x -> x` comes out as `a -> a`
3. We can generalize this type to `forall a . a -> a`
4. When we apply id the first time, we instantiate this polymorphic type with `Int`
5. When we apply id the second time, we instantiate this polymorphic type with `Int ->Int`

7.3.1 Type System 3

Types:

```
-- Mono-types
T ::= Int
    | T1 -> T2
    | a             -- type variables

-- Poly-types
S ::= T             -- mono
```

(continues on next page)

(continued from previous page)

```
    | forall a . S   -- polymorphic

-- where a  TVar, T  Type, S  Poly
```

7.3.1.1 Type Environment

The type environment now maps variables to poly-types: `G : Var -> Poly`

- example, `G = [z:  Int, id:  forall a . a -> a]`

7.3.1.2 Type Substitutions

We need a mechanism for replacing all type variables in a type with another type:

A type substitution is a finite map from type variables to types: `U : TVar -> Type`

- example: `U1 = [a / Int, b / (c -> c)]`

To apply a substitution U to a type T means replace all type vars in T with whatever they are mapped to in U

- example 1: `U1 (a -> a) = Int -> Int`
- example 2: `U1 Int = Int`

7.3.1.3 Typing Rules

We need to change the typing rules so that:

```
-- 1. variables and their definitions can have polymorphic types
[T-Var] G |- x :: S           if x:S in G

        G |- e1 :: S   G, x:S |- e2 :: T
[T-Let] -------------------------------------
           G |- let x = e1 in e2 :: T

-- 2. we can instantiate a type scheme into a type
         G |- e :: forall a . S
[T-Inst] -----------------------
          G |- e :: [a / T] S

-- 3. we can generalize a type with free type variables into a type scheme
             G |- e :: S
[T-Gen] ----------------------- if not (a in FTV(G))  -- FTV = Free Type Variables
        G |- e :: forall a . S

-- the rest of the rules are the same:
[T-Num] G |- n :: Int

        G |- e1 :: Int    G |- e2 :: Int
[T-Add] ---------------------------------
            G |- e1 + e2 :: Int

           G,x:T1 |- e :: T2
[T-Abs] -------------------------
        G |- \x -> e :: T1 -> T2
```

(continues on next page)

(continued from previous page)

```
        G |- e1 :: T1 -> T2    G |- e2 :: T1
[T-App] ------------------------------------  -- modus ponens!
                G |- e1 e2 :: T2
```

7.3.1.4 Examples

```
-- derive: [] |- \x -> x :: forall a . a -> a

[T-Var] ---------------
        [x:a] |- x :: a
[T-Abs] -----------------------
        [] |- \x -> x :: a -> a
[T-Gen] ----------------------------------  not (a in FTV([]))
        [] |- \x -> x :: forall a . a -> a

-- derive: [x:a] |- x :: forall a . a
-- not derivable, since a is not in FTV([x:a])

-- derive: G1 |- id 5 :: Int where G1 = [id : (forall a . a -> a)]

[T-Var] -----------------------------
        G1 |- id :: forall a . a -> a
[T-Inst]-----------------------         -------------- [T-Num]
        G1 |- id :: Int -> Int          G1 |- 5 :: Int
[T-App] ----------------------------------------------
        G1 |- id 5 :: Int

-- see slides page 12 for example 3
```

7.3.2 Representing Types

The eventual goal is to create a function `infer`, which:

- given a context G and an expression e,
- returns a type T s.t. `G |- e :: T`
- or reports a type error

```
data Type = TInt       -- int
    | Type :=> Type    -- T1 -> T2
    | Var String       -- a, b, c

data Poly = Mono Type
    | Forall TVar Poly

type TVar = String
type TEnv = [(Id, Poly)]  -- type environment
type Subst = [(String, Type)] -- type sub
```

Main idea: let's implement infer like this:

1. Depending on the kind of expression, find the typing rule that applies to it
2. If the rule has premises, recursively call `infer` to obtain the types of subexpressions

3. Combine the types of subexpressions according to the conclusion of the rule
4. If no rule applies, report a type error

```
-- | This is not the final version!!!
infer :: TypeEnv -> Expr -> Type
infer _    (ENum _)      = TInt
infer tEnv (EVar var)    = lookup var tEnv
infer tEnv (EAdd e1 e2) =
    if t1 == TInt && t2 == TInt
        then return TInt
        else throw "type error: + expects Int operands"
    where
        t1 = infer tEnv e1
        t2 = infer tEnv e2
```

The problem is, some of our typing rules are nondeterministic (see slides pg. 13)

1. guessing type

```
infer tEnv (ELam x e) = tX :=> tBody
    where
        tEnv' = extendTEnv x tX tEnv
        tX    = ???          -- ??????
        tBody = infer tEnv' e
```

2. guessing when to generalize

solution:

1. whenever we need to guess a type, don't. just return a fresh type variable
2. whenever a rule imposes a constraint on a type, try to find the right substitution for the free type vars to satisfy the constraint (unification)

7.3.3 Unification

The unification problem: given two types T1 and T2, find a type substitution U s.t. `U T1 = U T2`.

Such a substitution is called a unifier of T1 and T2.

e.g.:

1. The unifier of `a` and `Int` is `[a/Int]`
2. `a -> a` and `Int -> Int` is `[a/Int]`
3. `a -> Int` and `Int -> b` is `[a/Int, b/Int]`
4. `Int` and `Int` is `[]`
5. `a` and `a` is `[]`
6. `Int` and `Int -> Int` is invalid
7. `Int` and `a -> a` is invalid
8. `a` and `a -> a` is invalid
9. `b` and `a -> a` is `[b/a -> a]`

7.3.3.1 Infer 2

To add constraint-based typing, we need to keep track of the current substitution:

```
-- | Now has to keep track of current substitution!
infer :: Subst -> TypeEnv -> Expr -> (Subst, Type)
infer sub _    (ENum _)     = (sub, TInt)
infer sub tEnv (EVar var)   = (sub, lookup var tEnv)

-- Lambda case: simply generate fresh type variable!
infer sub tEnv (ELam x e) = (sub1, tX' :=> tBody)
    where
        tEnv'          = extendTEnv x tX tEnv
        tX             = freshTV -- we'll get to this
        (sub1, tBody)  = infer sub tEnv' e
        tX'            = apply sub1 tX

-- Add case: recursively infer types of operands
-- and enforce constraint that they are both Int
infer sub tEnv (EAdd e1 e2) = (sub4, TInt)
    where
        (sub1, t1) = infer sub tEnv e1    -- 1. infer type of e1
        sub2       = unify sub1 t1 Int    -- 2. constraint: t1 is Int
        tEnv'      = apply sub2 tEnv      -- 3. apply subst to context (sets in scope)
        (sub3, t2) = infer sub2 tEnv' e2 -- 4. infer e2 type in new ctx
        sub4       = unify sub3 t2 Int    -- 5. constraint: t2 is Int
```

Note: Fresh Type Variables

How do you create a new fresh type variable every time? You'll have to pass an argument along.

7.3.4 Polymorphism

When do we generalize a type like `a -> a` to `forall a . a -> a`?

When do we instantiate a polymorphic type and to what?

Generalization and Instantiation

- **Whenever we infer a type for a let-defined variable, generalize it**
    - It's safe, even when not necessary
- Whenever we see a variable with polymorphic type, instantiate it with a fresh type variable

CHAPTER

EIGHT

TYPE CLASSES

Slides

8.1 Quizzes

cse116-plus-type-ind -> E

cse116-ord-ind -> C

cse116-read-ind -> A

8.2 Intro

Let's think about overloading operators - `1 + 1` and `1.0 + 1.1` work slightly differently

This is **ad-hoc overloading** - to compare/add values of multiple types

Note: Haskell has no caste system, so functions are first-class citizens; what class are operators then?

8.3 Qualified Types

```
:type (+)
(+) :: (Num a) => a -> a -> a
```

+ takes in any class that is an instance of or implements `Num` - Num is a predicate/constraint

A **typeclass** is a collection of operations that must exist for the underlying type.

8.3.1 Eq

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

A type a is an instance of `Eq` if these operations exist on it.

8.4 Creating Instances

```
data Unshowable = A | B | C

instance Eq Unshowable where
    (==) A A = True
    (==) B B = True
    (==) C C = True
    (==) _ _ = False
    (/=) x y = not (x == y)
```

8.5 Automatic Derivation

```
data Showable = A' | B' | C'
    deriving (Eq, Show)
```

Haskell can automatically generate instances!

8.6 Standard Typeclass Hierarchy

```
class (Eq a, Show a) => Num a where  -- all Nums must derive from Eq and Show
    (+) :: a -> a -> a
    ...
```

8.7 Using Typeclasses

Let's build a small lib for environments mapping keys to values:

```
data Env k v
    = Def v  -- default
    | Bind k v (Env k v)  -- bind k to v, recursive structure
    deriving (Show)

-- API:
-- >>> let env0 = add "cat" 10.0 (add "dog" 20.0 (Def 0))

-- >>> get "cat" env0
-- 10

-- >>> get "dog" env0
-- 20

-- >>> get "horse" env0
-- 0

-- implementation:
add :: k -> v -> Env k v -> Env k v
add key val env = Bind key val env
```

(continues on next page)

(continued from previous page)

```
get :: (Eq k) => k -> Env k v -> v  -- note that k has to derive Eq!
get key (Def v)           = v
get key (Bind ek ev env) | k == ek    = ev
                         | otherwise = get key env
```

What about an optimized version that stores keys in increasing order, to optimize add and get?

1. the types of get and add: `get :: (Ord k) => k -> Env k v -> v` need to add Ord
2. the type of Env: move the default so that we don't have to recurse to the end

8.8 Explicit Signatures

In some cases using typeclasses, explicit signatures are required:

e.g. `read :: (Read a) => String -> a`, the opposite of `Show`

We have to do: `(read "2") :: Int` or `(read "2") :: Float`

CHAPTER

NINE

MONADS

9.1 Abstracting Code Patterns

Recall: the *Map* HOF works on lists

What if we wanted to, for example, show all elements of a tree?

```
mapList :: (a -> b) -> List a -> List b
mapTree :: (a -> b) -> Tree a -> Tree b
gmap    :: (Mappable t) => (a -> b) -> t a -> t b

class Functor where
    fmap :: (a -> b) -> t a -> t b

instance Functor [] where
    fmap = mapList

instance Functor Tree where
    fmap = mapList
```

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

O