Haskell, Continued

Announcements

- I'll post HW6 soon, due December ~~3~~ 1
  - Please install GHC as soon as possible
  - Post on Submitty forum if you hit a snag
  - Thanks Caleb and Sam for the useful tips

  - Work on exercises from Lectures 19 and 20 to get started programming in Haskell

Lecture Outline

- *Haskell*
  - *Covered basic syntax, types and functions*

  - *Interpreters for the lambda calculus*
  - *Lazy evaluation*
  - *Static typing and static type inference*
  - *Algebraic data types and pattern matching*
  - *Higher-order functions (next time)*
  - *Type classes (next time)*
  - *Monads … and more (next time)*

Interpreters for the Lambda Calculus

- An interpreter for the lambda calculus is a program that reduces lambda expressions to "answers"

- We must specify WHNF, HNF, NF
  - Definition of "answer". Which normal form?
  - Reduction strategy. How do we choose redexes in an expression? APPLICATIVE ORDER or NORMAL ORDER

An Interpreter

Haskell syntax:
let …. in
case f of
→

- Definition by cases on $\mathbf{E} ::= \mathbf{x} \mid (\lambda \mathbf{x}.\ \mathbf{E}_1) \mid (\mathbf{E}_1\ \mathbf{E}_2)$

$\text{interpret}(\mathbf{x}) = \mathbf{x}$

$\text{interpret}(\lambda \mathbf{x}.\mathbf{E}_1) = \lambda \mathbf{x}.\mathbf{E}_1$ ← Hint you are in WHNF

$\text{interpret}(\mathbf{E}_1\ \mathbf{E}_2) = \text{let } \mathbf{f} = \text{interpret}(\mathbf{E}_1)$

(fun: $\mathbf{E}_1$, arg: $\mathbf{E}_2$)

in case f of

$\lambda \mathbf{x}.\mathbf{E}_3 \to \text{interpret}(\mathbf{E}_3[\mathbf{E}_2/\mathbf{x}])$

- $\to \mathbf{f}\ \mathbf{E}_2$ ← Hint it's in WHNF

Apply function before "interpreting" the argument

Hint for Normal order

- What normal form: ? WHNF
- What strategy: ? NORMAL ORDER

Another Interpreter

- Definition by cases on $\mathbf{E} ::= \mathbf{x} \mid (\lambda\mathbf{x}.\ \mathbf{E}_1) \mid (\mathbf{E}_1\ \mathbf{E}_2)$

interpret($\mathbf{x}$) = $\mathbf{x}$

interpret($\lambda\mathbf{x}.\mathbf{E}_1$) = $\lambda\mathbf{x}.\mathbf{E}_1$

interpret($\mathbf{E}_1\ \mathbf{E}_2$) = let **f** = interpret($\mathbf{E}_1$)

a = interpret($\mathbf{E}_2$)

in case **f** of

$\lambda\mathbf{x}.\mathbf{E}_3$ -> interpret($\mathbf{E}_3[\mathbf{a}/\mathbf{x}]$)

\- -> **f a**

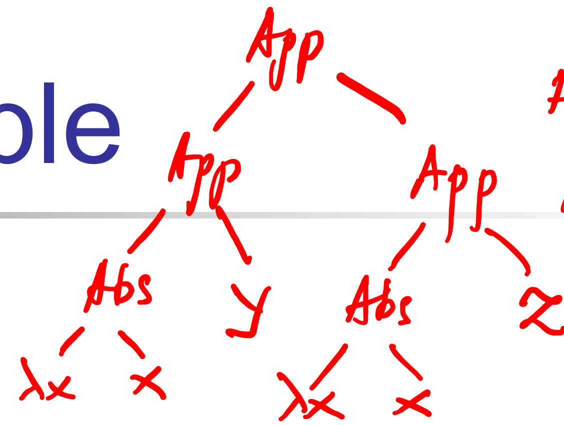
- What normal form: ? WHNF
- What strategy: ? Applicative order

Interpreter Example

WHNF, NORMAL ORDER

- $(\lambda x.x)\ y\ ((\lambda x.x)\ z)$

E_1 E_2 E_3



FREE VARS: {y, z}

REDEXES:

$(\lambda x.x)\ z$

$(\lambda x.x)\ y$

interpret $(((\lambda x.x)\ y)\ ((\lambda x.x)\ z))$ -- App E_1 E_2

$f \leftarrow$ interpret $((\lambda x.x)\ y)$ -- App E_1 E_2

$f' \leftarrow$ interpret $(\lambda x.x)$

returns $\lambda x.x$

return interpret $(y) = y$ -- matches $\lambda x.E_3$ case

return $y\ ((\lambda x.x)\ z)$ -- matches - case

Homework

- Step-by-step Normal order to Normal form interpretation

An Aside: Exam Question

- **P = λf.λs.λb.b f s**
- **F = λp.p (λx.λy.x)** **S = λp.p (λx.λy.y)**
- Reduce **S (P v w)** via applicative order reduction

$S\ ((\lambda f.\lambda s.\lambda b.b\ f\ s)\ v\ w) \rightarrow$

$S\ ((\lambda s.\lambda b.b\ v\ s)\ w) \rightarrow_\beta$

$S\ (\lambda b.b\ v\ w) =$

$(\lambda p.p\ (\lambda x.\lambda y.y))\ (\lambda b.b\ v\ w) \rightarrow_\beta$

$(\lambda b.b\ v\ w)\ (\lambda x.\lambda y.y) \rightarrow_\beta (\lambda x.\lambda y.y)\ v\ w \rightarrow_\beta$

$(\lambda y.y)\ w \rightarrow \boxed{w}$

An Aside: Exam Question

- **P = λf.λs.λb.b f s**
- **F = λp.p (λx.λy.x)** **S = λp.p (λx.λy.y)**
- Reduce **S (P v w)** via normal order reduction

Lazy Evaluation

- Unlike Scheme (and most programming languages) Haskell does use lazy evaluation, i.e., normal order reduction
  - It won't evaluate an expression until it is needed

> f x y = x*y

> f (5+1) (5+2)

--- evaluates to **(5+1)** * **(5+2)**

--- evaluates argument when needed

Lazy Evaluation

- In Scheme:

```
(define (fun x y) (* x y))
> (fun (+ 5 1) (+ 5 2)) -> (fun 6 7) -> (* 6 7)
                              -> 42

(define (fun n)
   (cons n (fun (+ n 1))))
> (car (fun 0))
>
```

Lazy Evaluation

: denotes "cons" : constructs a list with head **n** and tail **fun(n+1)**

- In Haskell:

fun n = n : fun(n+1)

> head (fun 0)

> 0

$P = \lambda f. \lambda s. \lambda b.\, b\, f\, s$ -- : (i.e. cons)

$F = \lambda p.\, p\, (\lambda x. \lambda y. x)$ -- head

$$(\lambda p.\, p\, (\lambda x.\lambda y.x))\ (fun\ 0) \rightarrow_\beta (fun\ 0)\ (\lambda x. \lambda y. x)$$

$$= (((\lambda f.\lambda s.\lambda b.\, b\, f\, s)\ 0)\ (fun\ (0+1)))\ (\lambda x.\lambda y.x)$$

$$= (\lambda f.\lambda s.\lambda b.\, b\, f\, s)\ 0\ (fun\ (0+1))\ (\lambda x.\lambda y.x)$$

$$\rightarrow_\beta^* (\lambda x.\lambda y.x)\ 0\ (fun\ (0+1)) \rightarrow_\beta^* 0$$

Lazy Evaluation

> take 2 (repeat 1)
[1, 1]

> f x = [] --- **f** takes **x** and returns the empty list

> f (repeat 1) --- **repeat** produces infinite list **[1,1…**

> []

> head ([1..]) --- **[1..]** is the infinite list of integers

> 1

> take 10 [1..]

[1,3..]

- Lazy evaluation allows infinite structures!

fun n = n : fun (n+2)

Aside: Python Generators

[1..] itertools

```
def gen(start):
    n = start
    while True:
      yield n
      n = n+1

gen_obj = gen(0)
print(next(gen_obj))   0
print(next(gen_obj))   1
print(next(gen_obj))   2
```

Lazy Evaluation

- Generate the (infinite) list of positive even integers

[2,4..]

map (\x-> x*2) [1..]

- Generate an (infinite) list of "fresh variables"

map (\x-> show x ++"_") [1..] => ["1_", "2_" ...]

Type conversion of int type to String

Lazy Evaluation

- Exercise: write a function that generates the (infinite) list of prime numbers

2 3 4 5 6 7 8 9 10 11 12 …

3 5 7 9 11 …

5 7 11 …

7 11 …

11 …

Static Typing and Type Inference

- Unlike Scheme, which is dynamically typed, Haskell is statically typed!
- Unlike Java/C++ we don't have to write type annotations. Haskell infers types!

[1..] or [True]

> let f x = head x in f ~~True~~

f :: [a] -> a   Bool

- Couldn't match expected type '[a]' with actual type 'Bool'
- In the first argument of 'f', namely 'True'

  In the expression: f True …

Static Typing and Type Inference

Applies n times f on x

- Higher-order function **apply_n f n x:**

> apply_n f n x = if n==0 then x else apply_n f (n-1) (f x)

apply_n :: (a -> a) -> Int -> a -> a
(f) (n) (x) (result)

> apply_n (+1) True 0

> apply_n (+1) 24 0

<interactive>:32:1: error:

• Could not deduce (Num Bool) arising from a use of ‘apply_n’

from the context: Num t2

bound by the inferred type of it :: Num t2 => t2

at <interactive>:32:1-22

• In the expression: apply_n (+ 1) True 0

In an equation for ‘it’: it = apply_n (+ 1) True 0

Lecture Outline

- *Haskell*
  - *Covered basic syntax, types and functions*

  - *Interpreters for the lambda calculus*
  - *Lazy evaluation*
  - *Static typing and static type inference*
  - *Algebraic data types and pattern matching*
  - *More on higher-order functions (next time)*
  - *Type classes (next time)*
  - *Monads … and more (next time)*

Algebraic Data Types

- Algebraic data types are tagged unions (aka sums) of products (aka records)

```
data Shape = Line Point Point
           | Triangle Point Point Point
           | Quad Point Point Point Point
```

union

Haskell keyword

the new type

new constructors (a.k.a. tags, disjuncts, summands)
Line is a binary constructor, Triangle is a ternary …

Algebraic Data Types

- Constructors create values of the data type

```
let
  l1::Shape
  l1 = Line e1 e2

  t1::Shape = Triangle e3 e4 e5
  q1::Shape = Quad e6 e7 e8 e9
in
```

…

Algebraic Data Types in Haskell Homework

- Defining a lambda expression

```
type Name = String
data Expr = Var Name
          | Lambda Name Expr
          | App Expr Expr
          deriving (Eq, Show)
```

> e1 = Var "x" // lambda calculus term **x**

> e2 = Lambda "x" e1 // term **λx.x**

Exercise: Define an ADT for PLAN Expressions (Scheme HW4)

```
type Name = String
data Expr = Var Name
          | Val Int
          | Mymul Expr Expr
          | Myadd Expr Expr
          | Mylet Name Expr Expr
          deriving (Eq, Show)
evaluate :: Expr -> [(Name,Int)] -> Int
evaluate e env = …
```

Pattern Matching

Type signature of anchorPnt: takes a Shape and returns a Point.

- Examine values of an algebraic data type

```
anchorPnt :: Shape -> Point
anchorPnt s = case s of
                Line     p1 p2 -> p1
                Triangle p3 p4 p5 -> p3
                Quad     p6 p7 p8 p9 -> p6
```

- Two points
  - Test: does the given value match this pattern?
  - Binding: if it matches, deconstruct it and bind pattern params to corresponding arguments

Pattern Matching

- Pattern matching “deconstructs” a term

```
> let h:t = "ana" in t
”na”

> let (x,y) = (10,”ana”) in x
10
```

Examples of Algebraic Data Types

Polymorphic types.
a is a type parameter!

data Bool = True | False

data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun

data List **a** = Nil | Cons **a** (List **a**)

data Tree **a** = Leaf **a** | Node (Tree **a**) (Tree **a**)

data Maybe **a** = Nothing | Just **a**

Maybe type denotes that result of computation can be **a** or Nothing. Maybe is a monad.

Type Constructor vs. Data Constructor

Bool and Day are nullary type constructors:

data Bool = True | False

data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun

E.g., x::Bool, y::Day

Maybe is a unary type constructor

data Maybe **a** = Nothing | Just **a**

E.g., s::Maybe Sheep, e::Maybe Expr

The End