CSC324 Principles of Programming Languages

Lecture 4

October 5/6, 2020

Announcement

- Ex3 unit tests are released on Markus (debrief next slide)
- Ex4 is due this Saturday
- P1 will be released Oct 5th evening. You can complete P1 after this lecture
- Week 5 (after reading week) will be P1 only
  - Lectures become office hours
  - Labs become office hours
  - More TA office hours posted in week 5 overview

Ex3 Debrief

Generally well done!

- Using `eval-calc` as a helper function to `foldl` is generally a bad idea
- Use a helper function instead
- Otherwise your code will not work for exercise 4
- Test cases are mostly about nesting `let*` expressions in various ways

Another common issue was actually... pattern matching!

Ex3 Debrief: Pattern Matching

- `(cons x xs)` is not the only pattern you can use!
- Example:
  - `(list '+ e1 e2)`
  - `(cons '+ (cons e1 (cons e2 '())))`
- If you want to match the symbol `let*`, you need to quote it
  - `(list 'let* e1 e2)`
  - `(list let* e1 e2)` – treats `let*` as any variable name

Overview

This week, we'll switch gears and talk more about **Haskell**

- Types and type systems
- Haskell's type system

Type systems is a very active area of programming languages research!

Wat

https://www.destroyallsoftware.com/talks/wat

Type Systems

Types Terminology

- **Type**: set of values, plus the allowable operations on those values
- **Type System**: set of rules governing the semantics of types
  - How types are defined
  - Syntax rules for conveying type information
  - How types affect the meaning of programs

Types

- Python: `1 + "hi"`
  - TypeError
- Racket: `(+ 1 "hi")`
  - contract violation
- Haskell: `1 + "hi"`
  - `No instance of (Num [Char]) arising from a use of '+'`
- JavaScript: `1 + "hi"`
  - No errors. “1hi”

In the chat. . .

What is the benefit to having a type system, and reasoning about types?

Definitions

Strong/Weak typing: does a value have a fixed type?

- **Strongly-typed** language: every value has a fixed type
- **Weakly-typed** language: values can be coerced at runtime to be of different types
- Strong/weak typing is a spectrum
  - e.g. If a language supports `3.5+4`, is it weakly-typed?

Static/Dynamic typing: when is type information checked/inferred?

- **Statically-typed** language: type information processed at **compile-time**, before the program runs
  - Often requires source code annotations (C, Java)
  - . . . but not always! (Think about your Haskell code.)
- **Dynamically-typed** language: no type information is checked until the program runs

Languages

Where does Haskell and Racket belong in this table?

What about Python, Java, C, C++, JavaScript?

	Dynamic	Static
Strong		
Weak		

Languages

	Dynamic	Static
Strong	Python, Racket	Java, Haskell
Weak	JavaScript	C/C++

Static Typing: Tradeoffs

Statically-checked languages like Haskell might reject correct programs

```
Prelude> (if True then 4 else "x") + 1
```

But these languages do guarantee that there are no type errors at runtime!

These languages can optimize code by tuning it for specific types

Racket's Type System

The function `member` can return two different types:

```
> (member 4 '(2 4 6 8))
'(4 6 8)
> (member 10 '(2 4 6 8))
#f
```

This behaviour isn't supported in Haskell: the type system has to know, for sure, the function type

Haskell's Type System

Displaying Types

Use `:type` or `:t` to display the type of an expression

```
> :t True            ; a boolean
Bool
> :t [True, True]    ; a list of booleans
[True, True] :: [Bool]
> :t [True, True, "hi"]
... type error
```

This `:t` is a REPL command, *not* a Haskell expression!

Haskell Function Types

```
> :t not
not :: Bool -> Bool
> :t (&&)
(&&) :: Bool -> Bool -> Bool
```

Haskell Function Types

```
> :t not
not :: Bool -> Bool
> :t (&&)
(&&) :: Bool -> Bool -> Bool
```

All functions are automatically curried, so the above type annotation is equivalent to

```
(&&) :: Bool -> (Bool -> Bool)
```

Haskell Function Types

```
> :t not
not :: Bool -> Bool
> :t (&&)
(&&) :: Bool -> Bool -> Bool
```

All functions are automatically curried, so the above type annotation is equivalent to

```
(&&) :: Bool -> (Bool -> Bool)
```

```
> :t (&&) True -- partially-apply &&
(&&) True :: Bool -> Bool
```

Breakout Group

Exercise 1 and 2 only.

Exercise 1. How does Haskell represent. . .

A list of Booleans

Exercise 1. How does Haskell represent...

A list of Booleans

- `[Bool]`

A function that takes a Boolean and a string, and returns a Boolean

Exercise 1. How does Haskell represent. . .

A list of Booleans

- `[Bool]`

A function that takes a Boolean and a string, and returns a Boolean

- `Bool -> String -> Bool`

A function that takes three list of characters, and returns a list of Booleans

Exercise 1. How does Haskell represent...

A list of Booleans

- `[Bool]`

A function that takes a Boolean and a string, and returns a Boolean

- `Bool -> String -> Bool`

A function that takes three list of characters, and returns a list of Booleans

- `[Char] -> [Char] -> [Char] -> [Bool]`

Exercise 2

```
f :: Int -> Int -> Int
```

```
g x = f (head x) (h False)
```

- h takes a boolean and needs to return an Int, so...

Exercise 2

```
f :: Int -> Int -> Int
```

```
g x = f (head x) (h False)
```

- `h` takes a boolean and needs to return an Int, so...

```
h :: Bool -> Int
```

Exercise 2

```
f :: Int -> Int -> Int
```

```
g x = f (head x) (h False)
```

- `h` takes a boolean and needs to return an Int, so...

```
h :: Bool -> Int
```

- g has a single parameter `x`, and `(head x)` is an Int
- The return type g is the same as the return type of `f`

Exercise 2

```
f :: Int -> Int -> Int
```

```
g x = f (head x) (h False)
```

- `h` takes a boolean and needs to return an Int, so...

```
h :: Bool -> Int
```

- `g` has a single parameter `x`, and `(head x)` is an Int
- The return type `g` is the same as the return type of `f`

```
g :: [Int] -> Int
```

Algebraic Data Types

Defining New Types

We've seen how to define new types in Haskell:

```haskell
-- from Exercise 3
data Expr = Number Float     -- ^ numeric literal
          | Add Expr Expr    -- ^ addition
          | Sub Expr Expr    -- ^ subtraction
          | Mul Expr Expr    -- ^ multiplication
          | Div Expr Expr    -- ^ division
          deriving (Show, Eq)
```

Q: What is the name of the new **type** that is created?

Defining New Types

We've seen how to define new types in Haskell:

```
-- from Exercise 3
data Expr = Number Float    -- ^ numeric literal
          | Add Expr Expr   -- ^ addition
          | Sub Expr Expr   -- ^ subtraction
          | Mul Expr Expr   -- ^ multiplication
          | Div Expr Expr   -- ^ division
          deriving (Show, Eq)
```

Q: What is the name of the new **type** that is created?

Q: What are the **value constructors** created?

Simpler Example

```
data Point = Point Float Float
```

- `Point` on the left is a **new type**
- `Point` on the right is a **value constructor**: a way to create a `Point` value
  - A value constructor is a function!

```
> :t Point -- Point function
Point :: Float -> Float -> Point -- Point type
```

Types, Value Constructors. . .

Easier to keep things straight by using a different constructor name:

```
data Point = MyPoint Float Float
```

Project 1 Types

Which of the following are types? Value constructors?

```
data Spreadsheet = Spreadsheet [Definition] [Column]
                 deriving (Show, Eq)

data Definition = Def String Expr
                deriving (Show, Eq)

data Column = ValCol String [Value]
            | ComputedCol String Expr
            deriving (Show, Eq)
```

Project 1 Types

Which of the following are types? Value constructors?

```
data Spreadsheet = Spreadsheet [Definition] [Column]
                 deriving (Show, Eq)

data Definition = Def String Expr
                deriving (Show, Eq)

data Column = ValCol String [Value]
            | ComputedCol String Expr
            deriving (Show, Eq)
```

What is the type of the value constructor `ComputedCol`?

Unions

Data types with multiple value constructors are called **unions**

```
data Shape
    = Circle Point Float      -- centre and radius
    | Rectangle Point Point   -- opposite corners
```

Q: What is the type name?

Unions

Data types with multiple value constructors are called **unions**

```
data Shape
    = Circle Point Float      -- centre and radius
    | Rectangle Point Point   -- opposite corners
```

Q: What is the type name?

Q: What are the value constructor names?

Unions

Data types with multiple value constructors are called **unions**

```
data Shape
    = Circle Point Float       -- centre and radius
    | Rectangle Point Point    -- opposite corners
```

Q: What is the type name?

Q: What are the value constructor names?

Q: What is the type of `Circle` ?

Algebraic Data Types

An algebraic data type is a data type that is created using value constructors and unions

```
data Shape
    = Circle Point Float      -- centre and radius
    | Rectangle Point Point   -- opposite corners
```

Working with Algebraic Data Types

Write a function to multiply each of a `Points` coordinates by `n`.

```
data Point = MyPoint Float Float
```

```
scale :: Point -> Float -> Point
```

How do we “extract” the coordinates of a point?

Working with Algebraic Data Types

Write a function to multiply each of a `Points` coordinates by `n`.

```
data Point = MyPoint Float Float
```

```
scale :: Point -> Float -> Point
```

How do we “extract” the coordinates of a point?

Pattern matching!

```
scale (MyPoint x y) n = _____
```

How do we “scale” the coordinates?

Working with Algebraic Data Types

Write a function to multiply each of a `Points` coordinates by `n`.

```
data Point = MyPoint Float Float

scale :: Point -> Float -> Point
```

How do we “extract” the coordinates of a point?

Pattern matching!

```
scale (MyPoint x y) n = _____
```

How do we “scale” the coordinates?

Create a new `Point`!

```
scale (MyPoint x y) n = MyPoint (n * x) (n * y)
```

Breakout Group

Exercise 3

Exercise 3

```haskell
data Point = MyPoint Float Float
data Shape
    = Circle Point Float        -- centre and radius
    | Rectangle Point Point  -- opposite corners
area :: Shape -> Float
```

Exercise 3

```
data Point = MyPoint Float Float
data Shape
    = Circle Point Float        -- centre and radius
    | Rectangle Point Point     -- opposite corners
area :: Shape -> Float

area (Circle (MyPoint x y) rad) = 3.14 * rad * rad
```

Exercise 3

```haskell
data Point = MyPoint Float Float
data Shape
    = Circle Point Float       -- centre and radius
    | Rectangle Point Point  -- opposite corners
area :: Shape -> Float

area (Circle (MyPoint x y) rad) = 3.14 * rad * rad

area (Rectangle
        (MyPoint x1 y1)
        (MyPoint x2 y2)) = abs ((x2 - x1) * (y2 - y1))
```

Type alias

You'll see this line of code in project 1:

```
type Env =  Map String Value
```

The `Env` is a **type alias**, and is a short hand for the type `Map String Value`

No new type is created in that line.

Generic Polymorphism

Haskell Lists

```
data BoolList = Empty
              | Cons Bool BoolList

data IntList = Empty
             | Cons Int IntList

data StringList = Empty
                | Cons String StringList
```

Polymorphic types

- Polymorphism
  - "poly" = "many"
  - "morphe" = "form"
- **Generic** (or **parametric**) **polymorphism**
  - ability for an entity to behave in the same way regardless of "input" or "contained" type
- Haskell's lists are **generically polymorphic**

Types of List Functions

What is the type of a function like `head`, then?

Types of List Functions

What is the type of a function like `head`, then?

```
> :t head
head :: [a] -> a
```

Here, `a` is a **type variable**

Type Variables

A **type variable** is an identifier that can be instantiated to any type

```
head :: [a] -> a
```

In words: `head` takes a list of some type `a` of elements, and returns a value of type `a`

Instantiating Type Variables

```
head [True, False, True]
```

The type variable a gets instantiated to `Bool` here.

Generically polymorphic values

```
[True, False, True] ++ []
["CSC324", "is", "the", "best"] ++ []
[3, 2, 4] ++ []
```

Generically polymorphic values

```
[True, False, True] ++ []
["CSC324", "is", "the", "best"] ++ []
[3, 2, 4] ++ []

[3, 2, 4] ++ (tail [False])
```

Exercise: Types of Functions

```
> map (* 10) [1, 2, 3]
[10,20,30]
> map not [True, False]
[False,True]
> map even [1, 2, 3]
[False,True,False]
```

What is the type of `map`?

Exercise: More types

What is a suitable type for `h`?

```
h lst = case lst of
          [x]         -> x
          (x:True:_) -> x
```

Types as Constraints

Why Types?

- Type Checking Help Prevent Bugs
- Constraining types of polymorphic functions constrains their implementations!

Generic Polymorphism: Constraints (1)

```
f :: a -> a
```

What are the possible implementations for `f`?

(Remember that `a` can be instantiated to *any* type!)

Generic Polymorphism: Constraints (2)

```
f :: a -> b -> a
```

What are the possible implementations for `f`?

Generic Polymorphism: Constraints (3)

```haskell
f :: a -> [a]
```

What are the possible implementations for `f`?

Generic Polymorphism: Constraints (4)

```
f :: [a] -> [a]
```

What are the possible implementations for `f`?

Generic Polymorphism: Constraints (5)

```
f :: a -> b
```

What are the possible implementations for `f`?

Impure Functions

Why do we not discuss these type constraints in imperative languages?

```
T f(T x) {
    deleteFiles();
    return x;
}
```