CSC324 Principles of Programming Languages

Lecture 9

November 13, 2019

Announcements

- All remaining exercise & assignment posted
- Assignment 2 office hours to be announced

Exercise 8

- Fairly well done
- Most common issue: returning a *list* instead of a *set* in the helper function

Today

Start the final chapter of the course!

- 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

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”

Strong vs. Weak Typing

- **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 vs. Dynamic Typing

When does a program check/infer type information?

- **Statically-typed** language: type information and typechecking are 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

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

Static Typing: Tradeoffs

Statically-checked languages like Haskell might reject correct programs

```
> (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

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

```
> :t [True, True]
[True, True] :: [Bool]
> :t [True, True, "hi"]
... type error
```

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
```

Sectioning

- Currying allows us to fix function parameters from the left
- **Sectioning** lets us fix the left *or* right parameter of a binary operator

```
> f = (4 /) -- fix first parameter
> f 8
0.5
> g = (/ 4) -- fix second parameter
> g 8
2.0
```

Type Inferencing Exercise

Suppose that `f` has the following signature:

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

What would the type signatures of `g` and `h` need to be, if `g` has the below definition?

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

Algebraic Data Types

Defining New Types

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

```
-- from Exercise 6
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, Ord)
```

Defining New Types

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

```
-- from Exercise 6
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, Ord)
```

Simpler example:

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

Types, Value Constructors

```
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
```

Exercise: Scaling a Point

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

Exercise: Scaling a Point

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

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

Unions

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

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

Exercise: what is the type name? What are the value constructor names?

Algeberaic Data Types

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

In C, we can use a combination of `structs` and `unions` to support algebraic data types

But we'll be responsible for much more in C than in Haskell

Algebraic Data Types in C

A `Point` is just a `struct` with `x` and `y` members.

```
struct Point {
    float x, y; };
```

Algebraic Data Types in C...

We might try to store a `Shape` as a `union`:

```
union Shape {
    struct Circle {
        struct Point centre;
        float radius;} circle;
    struct Rectangle {
        struct Point corner1, corner2; } rectangle;
};
```

But we have no way to check whether it's currently storing a `circle` or `rectangle`!

Algebraic Data Types in C...

Solution: use a tag field.

```
struct Shape {
    enum {CIRCLE, RECTANGLE} shape_tag;
    union {
        struct Circle {
            ...
```

We'll stop here. Check the notes for more!

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 g?

```
apply x f = f x
funcs     = [(>=), (<)]
g         = map (apply 4) funcs
```

Exercise: Even more types

What is a suitable type for `h`?

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

Type Constructors

- A list is *not* a type
  - A list of `Bool` is a type, a list of `Char` is a type, etc.
- A list is a **type constructor** that requires one parameter in order to produce a type

The Type Constructor Maybe

- `Maybe` is another example of a type constructor
- Like lists, `Maybe` requires one parameter in order to produce a type
- Its two value constructors are `Nothing` and `Just`

```
> :t Nothing
Nothing :: Maybe a
> :t Just
Just :: a -> Maybe a
> Nothing
Nothing
> Just 5
Just 5
> if even 5 then Just 5 else Nothing
Nothing -- what is Nothing's type here?
```

Exercise: Types with Maybes

What is a suitable type for `mystery`?

```
mystery (Just _) = Just True
mystery Nothing = Just False
```

Exercise: More Types with Maybes

What is a suitable type for `mystery2`?

```
mystery5 f Nothing = Nothing
mystery5 f (Just x) = Just (f x)
```

Defining Type Constructors

- To define a type constructor, use a type variable in a `data` type declaration.

```
data Maybe a = Nothing
             | Just a
```

Polymorphism in Java

```
class ArrayList<T> { // generic in type T
  ...
}

public static void main(String[] args) {
  ArrayList<Integer> ints = new ArrayList<Integer>();
}
```

Exercise 9

Background

Exercise 9 is a variation of a problem that I had to solve for my startup:

- There is a spreadsheet or a data table of some sort
- Each column is annotated with the type “number” or “string”
- Some columns are computed based on the value of other columns using formulas

Task: type check the formulas to detect type errors

Data Types

```haskell
data Type = NumCol | StrCol deriving (Eq, Show)

type TypeEnv = Map.Map String Type

type Formulas = Map.Map String (Maybe Formula)
```

Possible Spreadsheet Formulas

```
data Formula = Column String
             | Plus   Formula Formula
             | Concat Formula Formula
             | Length Formula
             | NumToString Formula
            deriving (Eq, Show)
```

Example (no type error)

first_name	last_name	full_name

type: StrCol
formula:
Nothing
	type: StrCol
	formula:
	Nothing
		type: StrCol
		formula: `Just (Concat`
		`(Column "first_name")`
		`(Column "last_name"))`

Example (type error)

first_name	age	name_age
type: StrCol formula: Nothing	type: NumCol formula: Nothing	type: StrCol formula: `Just (Concat (Column "first_name") (Column "age"))`

Exercise 10

Continuation passing style

- In a regular program, the continuation of each expression is determined dynamically and implicitly by the interpreter implementation.
- **Continuation Passing Style** makes the control flow explicit

Function in Direct Style

This is an example of a function in direct style

```
add1 :: Float -> Float
add1 x = x + 1
```

Function in Direct Style

This is an example of a function in direct style

```
add1 :: Float -> Float
add1 x = x + 1
```

This is the same function written in continuation passing style:

```
cpsAdd1 :: Float -> (Float -> r) -> r
cpsAdd1 x k = k (x + 1)
```

Unpacking the CPS

```
cpsAdd1 :: Float -> (Float -> r) -> r
cpsAdd1 x k = k (x + 1)
```

The parameter `k` is the *continuation*.

(Web programmers: this programming style might remind you of *callbacks*!)

Another example: `insert`

The function `insert` inserts an element into a sorted list:

```
insert :: [Int] -> Int -> [Int]
insert []     y = [y]
insert (x:xs) y = if x > y
    then y:x:xs
    else x:(insert xs y)
```

CPS transforming the `insert` function

```
cpsInsert :: [Int] -> Int -> ([Int] -> r) -> r
cpsInsert []     y k = k [y]
cpsInsert (x:xs) y k = if x > y
    then k (y:x:xs)
    else cpsInsert xs y $ \res -> k (x:res)
```

CPS Rules

- Easier to discover them via examples
- Assignment 1 has some more examples/rules that you could use
- Assignment 1 involves programmatically performing CPS transformations

Other resources:

- http://matt.might.net/articles/by-example-continuation-passing-style/
- https://medium.com/@b.essiambre/continuation-passing-style-patterns-for-javascript-5528449d3070