CSE 114A: Fall 2021

Foundations of Programming Languages

Datatypes and Recursion

Owen Arden
UC Santa Cruz

Based on course materials developed by Nadia Polikarpova

What is Haskell?

- **Last week:**
  - built-in *data types*
    - base types, tuples, lists (and strings)
  - writing functions using pattern matching and recursion
- **This week:**
  - user-defined *data types*
    - and how to manipulate them using pattern matching and recursion
  - more details about recursion

Representing complex data

- **We've seen:**
  - *base* types: `Bool`, `Int`, `Integer`, `Float`
  - some ways to *build up* types: given types `T1, T2`
    - functions: `T1 -> T2`
    - tuples: `(T1, T2)`
    - lists: `[T1]`
- **Algebraic Data Types:** a single, powerful technique for building up types to represent complex data
  - lets you define your own data types
  - subsumes tuples and lists!

Product types

- Tuples can do the job but there are two problems…

```
deadlineDate :: (Int, Int, Int)
deadlineDate = (2, 4, 2019)

deadlineTime :: (Int, Int, Int)
deadlineTime = (11, 59, 59)

-- | Deadline date extended by one day
extension :: (Int, Int, Int) -> (Int, Int, Int)
extension = ...
```

- Can you spot them?

1. Verbose and unreadable

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

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

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

-- | Deadline date extended by one day
extension :: Date -> Date
extension = ...
```

A **type synonym** for T: a name that can be used interchangeably with T

2. Unsafe

- We want this to fail at compile time!!!
  `extension deadlineTime`
- *Solution*: construct two different **datatypes**

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

deadlineDate :: Date
deadlineDate = Date 2 4 2019

deadlineTime :: Time
deadlineTime = Time 11 59 59
```

Record Syntax

- Haskell's **record syntax** allows you to *name* the constructor parameters:
- Instead of

```
data Date = Date Int Int Int
```

- You can write:

```
data Date = Date {
  month :: Int,
  day   :: Int,
  year  :: Int
}
deadlineDate = Date 1 1 2019
deadlineMonth = month deadlineDate
```

Use the *field name* as a function to access part of the data

Building data types

- Three key ways to build complex types/values:
  1. **Product types** (**each-of**): a value of `T` contains a value of `T1` *and* a value of `T2` **[done]**
  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 a *sub-value* of the same type `Ts`

Example: NanoMD

- Suppose I want to represent a *text document* with simple markup. Each paragraph is either:
  - plain text (`String`)
  - heading: level and text (`Int` and `String`)
  - list: ordered? and items (`Bool` and `[String]`)
- I want to store all paragraphs in a *list*

```
doc = [ (1, "Notes from 130")            -- Lvl 1 heading
      , "There are two types of languages:" -- Plain text
      , (True, ["purely functional", "purely evil"])
                                          --^^ Ordered list
      ] -- But this doesn't type check!!!
```

Sum Types

- Solution: construct a new type for paragraphs that is a *sum* (*one-of*) the three options!
  - plain text (`String`)
  - heading: level and text (`Int` and `String`)
  - list: ordered? and items (`Bool` and `[String]`)
- I want to store all paragraphs in a *list*

```
data Paragraph =
    Text String         -- 3 constructors,
  | Heading Int String  -- each with different
  | List Bool [String]  -- parameters
```


Constructing datatypes

```
data T =
    C1 T11 .. T1k
  | C2 T21 .. T2l
  | ..
  | Cn Tn1 .. Tnm
```

`T` is the new **datatype**

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

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`

Constructing datatypes

You can think of a `T` value as a **box**:

- *either* a box labeled `C1` with values of types `T11 .. T1k` inside
- *or* a box labeled `C2` with values of types `T21 .. T2l` inside
- *or* ...
- *or* a box labeled `Cn` with values of types `Tn1 .. Tnm` inside

Apply a constructor = pack some values into a box (and label it)

- `Text "Hey there!"`
  - put `"Hey there!"` in a box labeled `Text`
- `Heading 1 "Introduction"`
  - put `1` and `"Introduction"` in a box labeled `Heading`
- Boxes have different labels but same type (`Paragraph`)

Example: NanoMD

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

Now I can create a document like so:

```
doc :: [Paragraph]
doc = [
    Heading 1 "Notes from 130"
  , Text "There are two types of languages:"
  , List True ["purely functional", "purely evil"]
  ]
```


Example: NanoMD

Now I want **convert documents in to HTML.**

I need to write a function:

```
html :: Paragraph -> String
html p = ??? -- depends on the kind of
paragraph!
```

How to tell what's in the box?

- Look at the label!

Pattern Matching

Pattern matching = looking at the label and extracting values from the box

- we've seen it before
- but now for arbitrary datatypes

```
html :: Paragraph -> String
html (Text str)        = ...
  -- It's a plain text! Get string
html (Heading lvl str) = ...
  -- It's a heading! Get level and string
html (List ord items)  = ...
  -- It's a list! Get ordered and items
```

Dangers of pattern matching (1)

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

What would GHCi say to:

```
html (Heading 1 "Introduction")
```

Answer: Runtime error (no matching pattern)

Dangers of pattern matching (1)

Beware of **missing** and **overlapped** patterns

- GHC warns you about *overlapped* patterns
- GHC warns you about *missing* patterns when called with `-W` (use `:set -W` in GHCi)

Pattern matching expression

We've seen: pattern matching in *equations*

You can also pattern-match *inside your program* using the **case** expression:

```
html :: Paragraph -> String
html p =
  case p of
    Text str -> unlines [open "p", str, close "p"]
    Heading lvl str -> ...
    List ord items -> ...
```

Pattern matching expression: typing

The **case** expression

```
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`
  - i.e. a variable or a constructor of `D` applied to other patterns

The expression `e` is called the *match scrutinee*

Building data types

- Three key ways to build complex types/values:
  1. **Product types** (**each-of**): a value of `T` contains a value of `T1` *and* a value of `T2` **[done]**
  2. **Sum types** (**one-of**): a value of `T` contains a value of `T1` *or* a value of `T2` **[done]**
  3. **Recursive types**: a value of `T` contains a *sub-value* of the same type `Ts`

Recursive types

Let's define **natural numbers** from scratch:

```
data Nat = ???
```

Recursive types

```
data Nat = Zero | Succ Nat
```

A `Nat` value is:

- either an *empty* box labeled `Zero`
- or a box labeled `Succ` with another `Nat` in it!

Some `Nat` values:

```
Zero                     -- 0
Succ Zero                -- 1
Succ (Succ Zero)         -- 2
Succ (Succ (Succ Zero))  -- 3
...
```


Functions on recursive types

Principle: Recursive code mirrors recursive data

1. Recursive type as a parameter

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

Step 1: add a pattern per constructor

```
toInt :: Nat -> Int
toInt Zero     = ... -- base case
toInt (Succ n) = ... -- inductive case
                     -- (recursive call goes here)
```

1. Recursive type as a parameter

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

Step 2: fill in base case

```
toInt :: Nat -> Int
toInt Zero     = 0   -- base case
toInt (Succ n) = ... -- inductive case
                     -- (recursive call goes here)
```


1. Recursive type as a parameter

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

Step 3: fill in inductive case using a recursive call:

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


2. Recursive type as a result

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

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

2. Putting the two together

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

add :: Nat -> Nat -> Nat
add Zero     m = m              -- base case
add (Succ n) m = Succ (add n m) -- inductive case

sub :: Nat -> Nat -> Nat
sub n        Zero     = n       -- base case 1
sub Zero     _        = Zero    -- base case 2
sub (Succ n) (Succ m) = sub n m -- inductive case
```


2. Putting the two together

Lessons learned:

- **Recursive code mirrors recursive data**
- With **multiple** arguments of a recursive type, which one should I recurse on?
- The name of the game is to pick the right **inductive strategy**!

```
data Nat = Zero     -- base constructor
add
add
add
sub
sub
sub Zero     _        = Zero    -- base case 2
sub (Succ n) (Succ m) = sub n m -- inductive case
```


Lists

Lists aren't built-in! They are an *algebraic data type* like any other:

```
data List = Nil          -- base constructor
          | Cons Int List -- inductive constructor
```

- List `[1, 2, 3]` is *represented* as `Cons 1 (Cons 2 (Cons 3 Nil))`
- Built-in list constructors `[]` and `(:)` are just fancy syntax for `Nil` and `Cons`

Functions on lists follow the same general strategy:

```
length :: List -> Int
length Nil         = 0              -- base case
length (Cons _ xs) = 1 + length xs  -- inductive case
```

Lists

What is the right *inductive strategy* for appending two lists?

```
append :: List -> List -> List
append ??? ??? = ???
```


Lists

What is the right *inductive strategy* for appending two lists?

```
append :: List -> List -> List
append Nil ys = ys
append ??? ??? = ???
```


Lists

What is the right *inductive strategy* for appending two lists?

```
append :: List -> List -> List
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Trees

Lists are *unary trees* with elements stored in the nodes:

```
1 - 2 - 3 - ()
data List = Nil | Cons Int List
```

How do we represent *binary trees* with elements stored in the nodes?

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


Trees

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

data Tree = Leaf | Node Int Tree Tree

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


Functions on trees

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

Binary trees

```
() - () - () - 1
   |    |    \ 2
   |    \ 3
   \ () - 4
        \ 5
data Tree = Leaf Int | Node Tree Tree

t12345 = Node
          (Node (Node (Leaf 1) (Leaf 2)) (Leaf 3))
          (Node (Leaf 4) (Leaf 5))
```


Example: Calculator

I want to implement an arithmetic calculator to evaluate expressions like:

- `4.0 + 2.9`
- `3.78 – 5.92`
- `(4.0 + 2.9) * (3.78 - 5.92)`

What is a Haskell datatype to *represent* these expressions?

```
data Expr = ???
```


Example: Calculator

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

How do we write a function to *evaluate* an expression?

```
eval :: Expr -> Float
```

Example: Calculator

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

How do we write a function to *evaluate* an expression?

```
eval :: Expr -> Float
eval (Num f)     = f
```


Example: Calculator

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

How do we write a function to *evaluate* an expression?

```
eval :: Expr -> Float
eval (Num f)     = f
eval (Add e1 e2) = eval e1 + eval e2
```


Example: Calculator

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

How do we write a function to *evaluate* an expression?

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

Example: Calculator

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

How do we write a function to *evaluate* an expression?

```
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
```


Recursion is…

Building solutions for *big problems* from solutions for *sub-problems*

- **Base case:** what is the *simplest version* of this problem and how do I solve it?
- **Inductive strategy:** how do I *break down* this problem into sub-problems?
- **Inductive case:** how do I solve the problem *given* the solutions for subproblems?

Why use Recursion?

1. Often far simpler and cleaner than loops
    - But not always…
2. Structure often forced by recursive data
3. Forces you to factor code into reusable units (recursive functions)

Why *not* use Recursion?

1. Slow
2. Can cause stack overflow

Example: factorial

```
fac :: Int -> Int
fac n
  | n <= 1    = 1
  | otherwise = n * fac (n - 1)
```

```
<fac 4>
  ==> <4 * <fac 3>>            -- recursively call `fact 3`
  ==> <4 * <3 * <fac 2>>>      --  recursively call `fact 2`
  ==> <4 * <3 * <2 * <fac 1>>>> --   recursively call `fact 1`
  ==> <4 * <3 * <2 * 1>>>      --   multiply 2 to result
  ==> <4 * <3 * 2>>            --  multiply 3 to result
  ==> <4 * 6>                  -- multiply 4 to result
  ==> 24
```


Example: factorial

```
<fac 4>
  ==> <4 * <fac 3>>            -- recursively call `fact 3`
  ==> <4 * <3 * <fac 2>>>      --  recursively call `fact 2`
  ==> <4 * <3 * <2 * <fac 1>>>> --   recursively call `fact 1`
  ==> <4 * <3 * <2 * 1>>>      --   multiply 2 to result
  ==> <4 * <3 * 2>>            --  multiply 3 to result
  ==> <4 * 6>                  -- multiply 4 to result
  ==> 24
```

Each *function call* `<>` allocates a frame on the *call stack*

- expensive
- the stack has a finite size

Can we do recursion without allocating stack frames?

Tail recursion

Recursive call is the *top-most* sub-expression in the function body

- i.e. no computations allowed on recursively returned value
- i.e. value returned by the recursive call == value returned by function

Tail recursive factorial

Let's write a 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)
```


Tail recursive factorial

```
loop acc n
    | n <= 1    = acc
    | otherwise = loop (acc * n) (n - 1)
```

```
<facTR 4>
  ==>    <<loop 1  4>> -- call loop 1 4
  ==>   <<<loop 4  3>>> -- rec call loop 4 3
  ==>  <<<<loop 12 2>>>> -- rec call loop 12 2
  ==> <<<<<loop 24 1>>>>> -- rec call loop 24 1
  ==> 24                  -- return result 24!
```

Each recursive call **directly** returns the result

- without further computation
- no need to remember what to do next!
- no need to store the "empty" stack frames!

Tail recursive factorial

Because the *compiler* can transform it into a *fast loop*

```
facTR n = loop 1 n
  where
    loop acc n
      | n <= 1    = acc
      | otherwise = loop (acc * n) (n - 1)

function facTR(n){
  var acc = 1;
  while (true) {
    if (n <= 1) { return acc ; }
    else        { acc = acc * n; n = n - 1; }
  }
}
```


Tail recursive factorial

```
function facTR(n){
  var acc = 1;
  while (true) {
    if (n <= 1) { return acc ; }
    else        { acc = acc * n; n = n - 1; }
  }
}
```

- Tail recursive calls can be optimized as a **loop**
  - no stack frames needed!
- Part of the language specification of most functional languages
  - compiler **guarantees** to optimize tail calls

That’s all folks!