COMP3141

Software System Design and Implementation

Functional Programming Practice

Curtis Millar
CSE, UNSW
Term 2 2021

Recap: What is this course?

Software must be high quality:
correct, safe and secure.

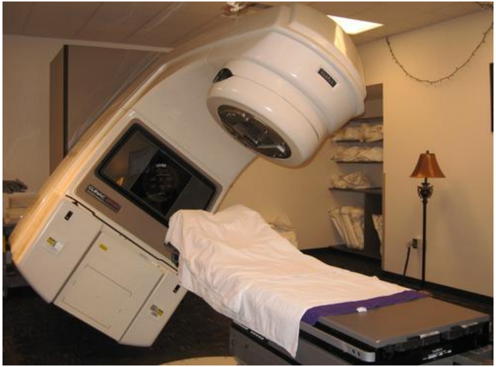
Software must developed
cheaply and quickly

Recall: Safety-critical Applications

For safety-critical applications, failure is not an option:

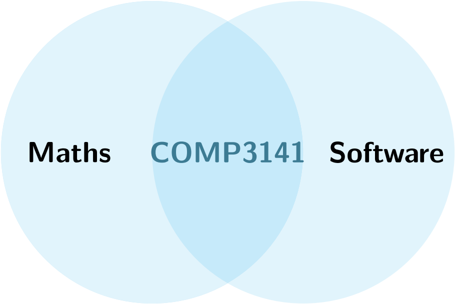
- planes, self-driving cars
- rockets, Mars probe
- drones, nuclear missiles
- banks, hedge funds, cryptocurrency exchanges
- radiation therapy machines, artificial cardiac pacemakers

Safety-critical Applications



A bug in the code controlling the Therac-25 radiation therapy machine was directly responsible for at least five patient deaths in the 1980s when it administered excessive quantities of beta radiation.

COMP3141: Functional Programming



Functional Programming: How does it Help?

1. Close to Maths: more abstract, less error-prone
2. Types: act as doc., the compiler eliminates many errors
3. Property-Based Testing: QuickCheck (in Week 3)
4. Verification: equational reasoning eases proofs (in Week 4)

COMP3141: Learning Outcomes

1. Identify basic Haskell **type errors** involving concrete types.
2. Work comfortably with **GHCi** on your working machine.
3. Use Haskell **syntax** such as guards, **let**-bindings, **where** blocks, **if** etc.
4. Understand the **precedence of function application** in Haskell, the `(.)` and `($)` operators.
5. Write Haskell programs to manipulate **lists** with recursion.
6. Makes use of **higher order functions** like *map* and *fold*.
7. Use λ-**abstraction** to define anonymous functions.
8. Write Haskell programs to compute **basic arithmetic**, **character, and string manipulation**.
9. Decompose problems using **bottom-up design.**

Functional Programming: History in Academia

1930s Alonzo Church developed lambda calculus (equiv. to Turing Machines)

1950s John McCarthy developed Lisp (LISt Processor, first FP language)

1960s Peter Landin developed ISWIM (If you See What I Mean, first pure FP language)

1970s John Backus developed FP (Functional Programming, higher-order functions, reasoning)

1970s Robin Milner and others developed ML (Meta-Language, first modern FP language, polymorphic types, type inference)

1980s David Turner developed Miranda (lazy, predecessor of Haskell)

1987- An international PL committee developed Haskell (named after the logician Curry Haskell)

... received Turing Awards (similar to Nobel prize in CS).
Functional programming is now taught at most CS departments.

Functional Programming: Influence In Industry

- Facebook's motto was:
  - "Move fast and break things."
  - as they expanded, they understood the importance of bug-free software
  - now Facebook uses functional programming!
- JaneStreet, Facebook, Google, Microsoft, Intel, Apple (... and the list goes on)
- Facebook building React and Reason, Apple pivoting to Swift, Google developing MapReduce.

Closer to Maths: Quicksort Example

Let's solve a problem to get some practice:

Example (Quicksort, recall from Algorithms)

Quicksort is a divide and conquer algorithm.

1. Picks a pivot from the array or list
2. Divides the array or list into two smaller sub-components: the smaller elements and the larger elements.
3. Recursively sorts the sub-components.

- What is the average complexity of Quicksort?
- What is the worst case complexity of Quicksort?
- Imperative programs describe **how** the program works.
- Functional programs describe **what** the program does.

Quicksort Example (Imperative)

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi - 1 do
        if A[j] < pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

Quick Sort Example (Functional)

```haskell
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
               where
                smaller = filter (\ a-> a <= x) xs
                larger  = filter (\ b-> b > x) xs
```

Is that it? Does this work?

Expressions and Types

Haskell constructs programs out of *expressions* but does not have any notion of a *statements* (unlike, e.g., C, Java, Python). Every expression has a type.
Let's practice figuring out the types of some expressions.

1. `True :: Bool`
2. `'a' :: Char`
3. `['a', 'b', 'c'] :: [Char]`
4. `"abc" :: [Char]`
5. `["abc"] :: [[Char]]`
6. `[('f',True), ('e', False)] :: [(Char, Bool)]`

1. In Haskell and GHCi using `:t`.
2. Using Haskell documentation and GHCi, answer the questions in this week's quiz (assessed!).

Functions

A function takes a single value as input and produces a single value as output.
Functions themselves are values, so a function can return a function that accepts further arguments.
We can create *anonymous* functions, i.e. functions without a name, using lambdas.
What is they type of the following lambda function?

```
\name -> "Hello, " ++ name ++ "!"
```

```
\name -> "Hello, " ++ name ++ "!" :: [Char] -> [Char]
```

Polymorphism

A function can have a *polymorphic* type, one that can take many forms. Such functions have *type variables* in the type that can be instantited with any type.
The most simply polymorphic function is the identity function that takes in input value and returns that same value with the same type.

```haskell
id = \x -> x :: forall a. a -> a
-- Alternatively
id :: a -> a
id x = x
```

For any type a, the function takes a value of that type and returns a value of the same type.

Polymorphism

Another surprisingly useful polymorphic function is the *constant* function that takes an input value and returns another function that, regardless of its input, always returns the value of the first argument. What is its type?

```haskell
const = \x -> \y -> x
```

```haskell
const :: a -> b -> a
const x y = x
```

Case and pattern matching

Some types have multiple *constructors*, such as `Char` (`'a'`, `'b'`, …), `Int` (…, `-1`, `0`, `1`, …), and `[a]` (`[]` and `x : xs`).
To access the values within a given datatype, we can *pattern match* a value of that type and *bind* parts of the constructor to names within a given expression.

```
howLong :: [a] -> String
howLong = \xs -> case xs of
        [] -> "empty"
        _ : [] -> "a single element"
        _ : xs -> "longer than a list that is " ++ (howLong xs)
```

Functions and pattern matching

```haskell
howLong :: [a] -> String
howLong = \xs -> case xs of
          []       -> "empty"
          _ : []   -> "a single element"
          _ : xs   -> "longer than a list that is " ++ (howLong xs)
```

Can also be written as:

```haskell
howLong :: [a] -> String
howLong []         = "empty"
howLong (_ : [])   = "a single element"
howLong (_ : xs)   = "longer than a list that is " ++ (howLong xs)
```

Higher Order Functions

As functions are also values, we can also accept functions as input values. How could we implement a function with the following type?

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

```haskell
flip :: (a -> b -> c) -> b -> a -> c
flip f y x = f x y
```

```haskell
flip :: (a -> b -> c) -> (b -> (a -> c))
flip f y x = (f x) y
```

Right Fold

Let's try and devise a more complex function for some practice.

Example (Demo Task)

We want a function that can consume an entire list and use each value in the list to construct a new value.

- This function should be able to take any operation and starting value.
- When this function is passed the operation `:` and the starting value `[]`, it should return the input list

```haskell
-- What is the type?
listFold = unimplemented

testListFold xs = listFold (:) [] xs == xs
```

Right Fold

Next we'll implement each of the following directly then using our fold function:

- `map`
- `reverse`
- `leftFold`
- `filter`
- `append`
- `concat`

Word Frequencies

Let's solve a problem to get some practice:

Example (First Demo Task)

Given a number n and a string s, generate a report (in `String` form) that lists the n most common words in the string s.

We must:

1. Break the input string into words.
2. Convert the words to lowercase.
3. Sort the words.
4. Count adjacent runs of the same word.
5. Sort by size of the run.
6. Take the first n runs in the sorted list.
7. Generate a report.

Numbers into Words

Let's solve a problem to get some practice:

Example (Demo Task)

Given a number n, such that $0 \leq n < 1000000$, generate words (in `String` form) that describes the number n.

We must:

1. Convert single-digit numbers into words ($0 \leq n < 10$).
2. Convert double-digit numbers into words ($0 \leq n < 100$).
3. Convert triple-digit numbers into words ($0 \leq n < 1000$).
4. Convert hexa-digit numbers into words ($0 \leq n < 1000000$).

Single Digit Numbers into Words

$$0 \leq n < 10$$

```
units :: [String]
units =
    ["zero", "one", "two", "three", "four", "five",
     "six", "seven", "eight", "nine", "ten"]
convert1 :: Int -> String
convert1 n = units !! n
```

Double Digit Numbers into Words

$$0 \leq n < 100$$

```haskell
teens :: [String]
teens =
    ["ten", "eleven", "twelve", "thirteen", "fourteen",
     "fifteen", "sixteen", "seventeen", "eighteen",
     "nineteen"]

tens :: [String]
tens =
    ["twenty", "thirty", "fourty", "fifty", "sixty",
     "seventy", "eighty", "ninety"]
```

Double Digit Numbers into Words Continued $(0 \leq n < 100)$

```haskell
digits2 :: Int -> (Int, Int)
digits2 n = (div n 10, mod n 10)
combine2 :: (Int, Int) -> String
combine2 (t, u)
    | t == 0              = convert1 u
    | t == 1              = teens !! u
    | t > 1 && u == 0     = tens !! (t-2)
    | t > 1 && u /= 0     = tens !! (t-2)
                            ++ "-" ++ convert1 u
convert2 :: Int -> String
convert2 = combine2 . digits2
```

Infix Notation

Instead of

```
digits2 n = (div n 10, mod n 10)
```

for **infix** notation, write:

```
digits2 n = (n `div` 10, n `mod` 10)
```

Note: this is not the same as single quote used for `Char` (`'a'`).

Simpler Guards but Order Matters

You could also simplify the guards as follows:

```
combine2 :: (Int, Int) -> String
combine2 (t,u)
  | t == 0       = convert1 u
  | t == 1       = teens !! u
  | u == 0       = tens !! (t-2)
  | otherwise    = tens !! (t-2) ++ "-" ++ convert1 u
```

but now the order in which we write the equations is crucial. `otherwise` is a synonym for `True`.

Where instead of Function Composition

Instead of implementing `convert2` as `digit2.combine2`, we can implement it directly using the `where` keyword:

```
convert2 :: Int -> String
convert2 n
  | t == 0        = convert1 u
  | t == 1        = teens !! u
  | u == 0        = tens !! (t-2)
  | otherwise     = tens !! (t-2) ++ "-" ++ convert1 u
  where (t, u) = (n `div` 10, n `mod` 10)
```

Triple Digit Numbers into Words ($0 \leq n < 1000$)

```haskell
convert3 :: Int -> String
convert3 n
    |  h == 0      = convert2 n
    |  t == 0      = convert1 h ++ "hundred"
    |  otherwise   = convert1 h ++ " hundred and "
                     ++ convert2 t
  where (h, t) = (n `div` 100, n `mod` 100)
```

Hexa Digit Numbers into Words
($0 \leq n < 1000000$)

```
convert6 :: Int -> String
convert6 n
    |  m == 0       = convert3 n
    |  h == 0       = convert3 m ++ "thousand"
    |  otherwise    = convert3 m ++ link h ++ convert3 h
    where (m, h) = (n `div` 1000, n `mod` 1000)
link :: Int -> String
link h = if (h<100) then " and " else " "
convert :: Int -> String
convert = convert6
```

Homework

1. Get Haskell working on your development environment. Instructions are on the course website.
2. Using Haskell documentation and GHCi, answer the questions in this week's quiz (assessed!).

COMP3141

Software System Design and Implementation

Introduction

Dr. Christine Rizkallah
UNSW
Term 2 2021

Who are we?

I am Dr. Christine Rizkallah, currently a lecturer at UNSW. I lead the Cogent team and my research specialty involves formal methods and programming languages for building reliable software.

Curtis Millar, is a UNSW academic who works on, among other things, projects involving trustworthy systems and formal methods.

Credits

We thank the former lecturers of this course for developing and continuously refining this course material. The course material offered greatly benefits from their contributions.

Dr. Liam O'Connor, who now a lecturer at the University of Edinburgh, and whose research involves programming languages for trustworthy systems.

Prof. Gabriele Keller, who now works at Utrecht University, and whose research expertise is on programming languages for formal methods and high performance computing.

Prof. Manuel Charkavarty, who now works at IOHK with a focus on research and development of programming languages and compilers for functional programming.

Contacting Us

`http://www.cse.unsw.edu.au/~cs3141`

Forum

There is a Discourse forum available on the website. Questions about course content should typically be made there. You can ask us private questions to avoid spoiling solutions to other students.

Administrative questions should be sent to `cs3141@cse.unsw.edu.au`.

What is this course?

Software must be high quality: **correct, safe and secure.**

Software must developed **cheaply and quickly**

Safety-uncritical Applications

;

Video games: Some bugs are acceptable, to save developer effort.

Safety-critical Applications

Remember a particularly painful uni group work assignment.

Now imagine you...

- Are travelling on a plane
- Are travelling in a self-driving car
- Are working on a Mars probe
- Have invested in a new hedge fund
- Are running a cryptocurrency exchange
- Are getting treatment from a radiation therapy machine
- Intend to launch some nuclear missiles at your enemies

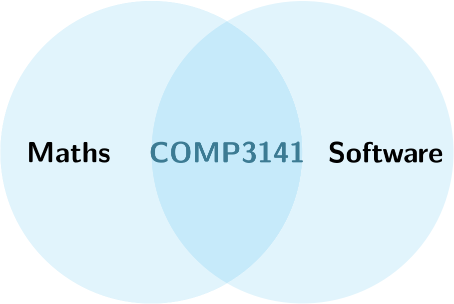
...running on software written by the person next to you on the participant list.

Safety-critical Applications

Airline Blames Bad Software in San Francisco Crash

The New York Times

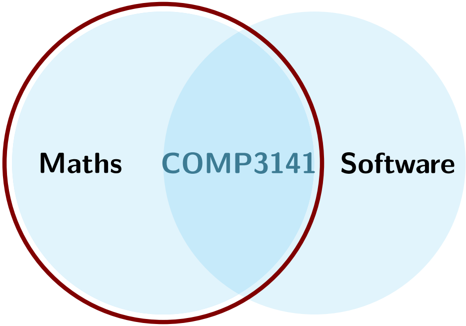
What is this course?



What is this course?

Maths?

- Logic
- Set Theory
- Proofs
- Induction
- Algebra (a bit)
- No calculus ☺

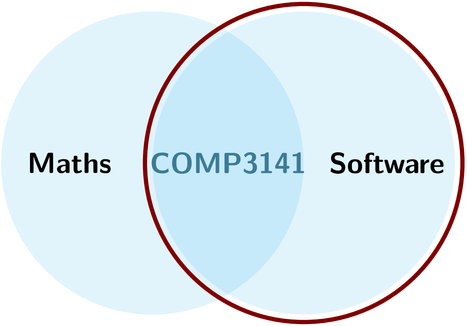


N.B: MATH1081 is neither necessary nor sufficient for COMP3141.

What is this course?

Software?

- Programming
- Reasoning
- Design
- Testing
- Types
- Haskell



N.B: Haskell knowledge is not a prerequisite for COMP3141.

What isn't this course?

This course is not:

- **a Haskell course**
- a verification course (for that, see COMP6721, COMP4161)
- an OOP software design course (see COMP2511, COMP1531)
- a programming languages course (see COMP3161).
- a WAM booster cakewalk (hopefully).
- a soul-destroying nightmare (hopefully).

Assessment

Warning

For many of you, this course will present a lot of new topics. Even if you are a seasoned programmer, you may have to learn as if from scratch.

- Class Mark (out of 100)
  - **Two** programming assignments, each worth 20 marks.
  - Weekly online quizzes, worth 20 marks.
  - Weekly programming exercises, worth 40 marks.
- Final Exam Mark (out of 100)
  - This includes 10 marks for active participation throughout the term. You will be provided with various methods and opportunities to demonstrate participation throughout the term.

$$result = \frac{class + exam}{2}$$

Lectures

- The lectures will be held on zoom. As there is a limit of 300 attendees on zoom, we will live stream the lectures to YouTube, in case we hit the zoom limit.
- I, Christine, or someone qualified whom I nominate, will run lectures introducing new material on Wednesdays.
- Curtis will reinforce this new material with questions and examples on Fridays.
- Curtis has kindly agreed to offer three help sessions throughout the term. The dates and times for these sessions will be announced at least one week in advance on the course website.
- You must attend lectures or watch recordings as they come out.
- Recordings will be available from the course website.
- Online quizzes are due one week after the lectures they examine, but do them early!

Books

There are no set textbooks for this course, however, there are various books that are useful for learning Haskell listed on the course website.
I can also provide more specialised text recommendations for specific topics.

Haskell

In this course we use Haskell, because it is the most widespread language with good support for mathematically structured programming.

“of type” Codomain

```haskell
f ::Int -> Bool
f x = (x > 0)
```

Function Name Domain

Input Output

In mathematics, we would apply a function by writing $f(x)$. In Haskell we write `f x`.

Demo: GHCi, basic functions

Currying

- In mathematics, we treat $\log_{10}(x)$ and $\log_2(x)$ and $\ln(x)$ as separate functions.
- In Haskell, we have a single function `logBase` that, given a number n, produces a function for $\log_n(x)$.

```
log10 :: Double -> Double
log10 = logBase 10

log2 :: Double -> Double
log2 = logBase 2

ln :: Double -> Double
ln = logBase 2.71828
```

What's the type of `logBase`?

Currying and Partial Application

```haskell
logBase :: Double -> (Double -> Double)
```

(parentheses optional above)

Function application associates to the **left** in Haskell, so:

`logBase 2 64` $\equiv$ `(logBase 2) 64`

Functions of more than one argument are usually written this way in Haskell, but it is possible to use tuples instead...

Tuples

Tuples are another way to take multiple inputs or produce multiple outputs:

```
toCartesian :: (Double, Double) -> (Double, Double)
toCartesian (r, theta) = (x, y)
   where x = r * cos theta
         y = r * sin theta
```

N.B: The order of bindings doesn't matter. Haskell functions have no side effects, they just return a result.

Higher Order Functions

In addition to returning functions, functions can take other functions as arguments:

```
twice :: (a -> a) -> (a -> a)
twice f a = f (f a)

double :: Int -> Int
double x = x * 2

quadruple :: Int -> Int
quadruple = twice double
```

Lists

Haskell makes extensive use of lists, constructed using square brackets. Each list element must be of the same type.

```
[True, False, True]    ::   [Bool]
[3, 2, 5+1]            ::   [Int]
[sin, cos]             ::   [Double -> Double]
[ (3,'a'),(4,'b') ]    ::   [(Int, Char)]
```

Map

A useful function is `map`, which, given a function, applies it to each element of a list:

```
map not [True, False, True] = [False, True, False]
map negate [3, -2, 4]       = [-3, 2, -4]
map (\x -> x + 1) [1, 2, 3] = [2, 3, 4]
```

The last example here uses a *lambda expression* to define a one-use function without giving it a name.

What's the type of map?

```
map :: (a -> b) -> [a] -> [b]
```

Strings

The type `String` in Haskell is just a list of characters:

```haskell
type String = [Char]
```

This is a *type synonym*, like a `typedef` in C.

Thus:

```haskell
"hi!" == ['h', 'i', '!']
```

Word Frequencies

Let's solve a problem to get some practice:

Example (First Demo Task)

Given a number n and a string s, generate a report (in `String` form) that lists the n most common words in the string s.

We must:

1. Break the input string into words.
2. Convert the words to lowercase.
3. Sort the words.
4. Count adjacent runs of the same word.
5. Sort by size of the run.
6. Take the first n runs in the sorted list.
7. Generate a report.

Function Composition

We used *function composition* to combine our functions together. The mathematical $(f \circ g)(x)$ is written `(f . g) x` in Haskell.

In Haskell, operators like function composition are themselves functions. You can define your own!

```
-- Vector addition
(.+) :: (Int, Int) -> (Int, Int) -> (Int, Int)
(x1, y1) .+ (x2, y2) = (x1 + x2, y1 + y2)
```

```
(2,3) .+ (1,1) == (3,4)
```

You could even have defined function composition yourself if it didn't already exist:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```

Lists

How were all of those list functions we just used implemented?

Lists are singly-linked lists in Haskell. The empty list is written as `[]` and a list node is written as `x : xs`. The value `x` is called the head and the rest of the list `xs` is called the tail. Thus:

```
"hi!" == ['h', 'i', '!'] == 'h':('i':('!':[]))
                         == 'h' : 'i' : '!' : []
```

When we define recursive functions on lists, we use the last form for pattern matching:

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

Equational Evaluation

```haskell
map f []     = []
map f (x:xs) = f x : map f xs
```

We can evaluate programs *equationally*:

```
map toUpper "hi!" ≡ map toUpper ('h':"i!")
                  ≡ toUpper 'h' : map toUpper "i!"
                  ≡ 'H' : map toUpper "i!"
                  ≡ 'H' : map toUpper ('i':"!")
                  ≡ 'H' : toUpper 'i' : map toUpper "!"
                  ≡ 'H' : 'I' : map toUpper "!"
                  ≡ 'H' : 'I' : map toUpper ('!':"")
                  ≡ 'H' : 'I' : '!' : map toUpper ""
                  ≡ 'H' : 'I' : '!' : map toUpper []
                  ≡ 'H' : 'I' : '!' : []
                  ≡ "HI!"
```

Higher Order Functions

The rest of this lecture will be spent introducing various list functions that are built into Haskell's standard library by way of live coding.

Functions to cover:

1. `map`
2. `filter`
3. `concat`
4. `sum`
5. `foldr`
6. `foldl`

In the process, we will introduce **let** and **case** syntax, **guards** and **if**, and the \$ operator.

Homework

1. Get Haskell working on your development environment. Instructions are on the course website.
2. Using Haskell documentation and GHCi, answer the questions in this week's quiz (assessed!).
3. Attend Curtis' online lecture on Friday!

COMP3141

Software System Design and Implementation

Induction, Data Types and Type Classes Practice

Curtis Millar
CSE, UNSW
11 June 2021

Product Types

```haskell
data Point = Point Float Float
           deriving (Show, Eq)

data Vector = Vector Float Float
            deriving (Show, Eq)

movePoint :: Point -> Vector -> Point
movePoint (Point x y) (Vector dx dy)
   = Point (x + dx) (y + dy)
```

Records

```haskell
data Colour = Colour { redC     :: Int
                     , greenC   :: Int
                     , blueC    :: Int
                     , opacityC :: Int
                     } deriving (Show, Eq)
```

Sum Types

```haskell
data LineStyle = Solid
               | Dashed
               | Dotted
               deriving (Show, Eq)

data FillStyle = SolidFill | NoFill
               deriving (Show, Eq)
```

Constructors

Constructors are how an value of a particular type is created.

```
data Bool = True | False
data Int = .. | -1 | 0 | 1 | 2 | 3 | ..
data Char = 'a' | 'b' | 'c' | 'd' | 'e' | ..
```

Custom Constructors

```
data PointType = PointConstructor Float Float
          deriving (Show, Eq)

data VectorType = VectorConstructor Float Float
           deriving (Show, Eq)
```

Here, `PointConstructor` and `VectorConstructor` are both constructors.
`PointConstructor` has the type `Float -> Float -> PointType` and
`VectorConstructor` has the type `Float -> Float -> VectorType`

Algebraic Data Types

Just as the `Point` constructor took two `Float` arguments, constructors for sum types can take parameters too, allowing us to model different kinds of shape:

```
data PictureObject
    = Path    [Point]      Colour LineStyle
    | Circle  Point Float  Colour LineStyle FillStyle
    | Polygon [Point]      Colour LineStyle FillStyle
    | Ellipse Point Float Float Float
              Colour LineStyle FillStyle
    deriving (Show, Eq)

type Picture = [PictureObject]
```

Here, `type` creates a *type alias* which provides only an alternate name that refers to an existing type.

Patterns in Function Definitions

1. Patterns are used to deconstruct an value of a particular type.
2. A pattern can be a binding to a hole (_), a name, or a constructor of the type.
3. When defining a function, each argument is bound using a separate pattern.

Patterns in Function Definitions

```
if' :: Bool -> a -> a -> a
if' True  then' _     = then'
if' False _     else' = else'
```

Patterns in Function Definitions

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Patterns in Function Definitions

```haskell
isVowel :: Char -> Bool
isVowel 'a' = True
isVowel 'e' = True
isVowel 'i' = True
isVowel 'o' = True
isVowel 'u' = True
isVowel _   = False
```

Records and Accessors

```haskell
data Colour = Colour { redC      :: Int, greenC   :: Int
                     , blueC     :: Int, opacityC :: Int
                     }

-- Is equivalent to

data Color = Color Int Int Int Int

redC     (Color r _ _ _) = r
greenC   (Color _ g _ _) = g
blueC    (Color _ _ b _) = b
opacityC (Color _ _ _ o) = o
```

Patterns in Expressions

```
factorial :: Int -> Int
factorial x =
  case x of
    0 -> 1
    n -> n * factorial (n - 1)
```

Newtype

`newtype` allows you to encapsulate an existing type to add constraints or properties without adding runtime overhead.

```
newtype Kilometers = Kilometers Float
newtype Miles = Miles Float

kilometersToMiles :: Kilometers -> Miles
kilometersToMiles (Kilometers kms) = Miles $ kms / 1.60934

milesToKilometers :: Miles -> Kilometers
milesToKilometers (Miles miles) = Kilometers $ miles * 1.60934
```

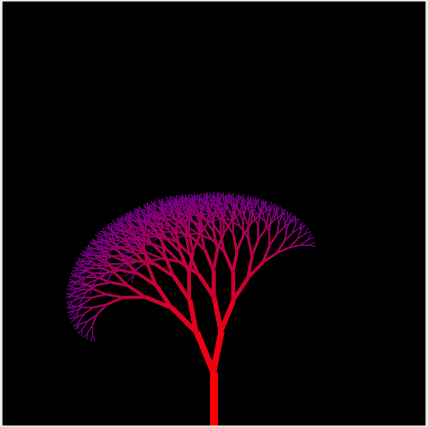
Natural Numbers

```haskell
data Nat = Zero
         | Succ Nat

add :: Nat -> Nat -> Nat
add Zero     n = n
add (Succ a) b = add a (Succ b)

zero = Zero
one = Succ Zero
two = add one one
```

1. `Nat` is recursive as it has the (Succ) constructor which takes a `Nat`.
2. `Nat` has the `Zero` constructor which does not recurse and acts like a *base case*.

More Cool Graphics

Example (Live Coding of Fractal Trees)



Type Classes

1. A type class has nothing to do with OOP classes or inheritance.
2. Type classes describe a set of behaviours that can be implemented for any type.
3. A function or type class instance can operate on a type variable constrained by a type class instead of a concrete type.
4. A type class is similar to an OOP interface.
5. When creating an instance of a type class with *laws*, you must ensure the laws are held manually (they cannot be checked by the compiler).
6. When using a type class with *laws* you can assume that all laws hold for all instances of the type class.

Show

`Show` simply allows us to take a type and represent it as a string.

Haskell Definition

```
class Show a where
      show :: a -> [Char]
```

This is implemented for all of the built-in types such as `Int`, `Bool`, and `Char`

Read

Effectively the 'dual' of `Show`, `Read` allows us to take a string representation of a value and decode it.

You can *think* of read as having the following definition, but it is actually somewhat more complex.

Definition

```
class Read a where
       read :: [Char] -> a
```

This is implemented for all of the built-in types such as `Int`, `Bool`, and `Char`

Ord

`Ord` allows us to compare two values of a type for a *partial* or *total inequality*.

Haskell Definition

```haskell
class Ord a where
        (<=) :: a -> a -> Bool
```

1. **Tranisitivity**: $x \leq y \land y \leq z \to x \leq z$
2. **Reflexivity**: $x \leq x$
3. **Antisymmetry**: $x \leq y \land y \leq x \to x = y$
4. **Totality** (total order): $x \leq y \lor y \leq x$

Eq

Eq allows us to compare two values of a type for an *equivalence* or *equality*.

Haskell Definition

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

1. **Reflexivity**: $x = x$
2. **Symmetry**: $x = y \to y = x$
3. **Tranisitivity**: $x = y \wedge y = z \to x = z$
4. **Negation** (equality): $x \neq y \to \neg(x = y)$
5. **Substitutivity** (equality): $x = y \to f\ x = f\ y$

Derived Instances

When defining a new type we can have the compiler generate instances of `Show`, `Read`, `Ord`, or `Eq` with the `deriving` statement at the end of the definition.

Haskell Example

```haskell
data Colour = Colour { redC      :: Int
                     , greenC    :: Int
                     , blueC     :: Int
                     , opacityC  :: Int
                     } deriving (Show, Eq)
```

Derived instances of `Ord` will be total orders and will order by fields in the order they appear in a product type and will order constructors in the same order they are defined. Derived instances of `Eq` will be strict equalities.

Kinds of Types

1. Just as values and functions in the *runtime language* of Haskell have *types*, types in the *type language* of Haskell have *kinds*.
2. The kind of a *concrete type* is `*`.
3. Just as *functions* exist over values (with the type `a -> b`), *type constructors* exist for types.
4. `* -> *` is a type constructor that takes a concrete type and produces a concrete type.

Maybe

Haskell Definition

```haskell
-- Maybe :: * -> *
data Maybe a = Just a
             | Nothing
-- Maybe Int :: *
```

1. `Maybe` is a type constructor that takes a type and produces a type that may or may not hold a value.
2. `Maybe Int` is a concrete type that may or may not hold an `Int`.

List

Haskell Definition

```haskell
-- List :: * -> *
data List a = Cons a (List a)
            | Nil

-- List Int :: *
```

1. `List a` is recursive as it has the (Cons) constructor which takes a `List a`.
2. `List a` has the `Nil` constructor which does not recurse and acts like a *base case*.
3. `List` is a type constructor that takes a type and produces a type that holds zero or more of a value.
4. `List Int` is a concrete type that zero or more values of type `Int`.

Haskell List

Definition

```haskell
-- [ ] :: * -> *
data [a] = a : (List a)
         | []
```

1. `[a, b, c]` is syntactic sugar for the constructor `(a : (b : (c : [])))`.
2. `"abc"` is syntactic sugar for the constructor `('a' : ('b' : ('c' : [])))`.
3. Both can also be used as patterns.

Tree

Haskell Definition

```haskell
-- Tree :: * -> *
data Tree a = Node a (Tree a) (Tree a)
            | Leaf
```

1. `Tree a` is recursive in the same manner as `List a`.
2. `Tree` is a type constructor that takes a type and produces a type that holds zero or more of a value in a tree.
3. `Tree Int` is a concrete type that holds zero or more values of type `Int` in a tree.

Semigroup

A *semigroup* is a pair of a set S and an operation $\bullet : S \to S \to S$ where the operation $\bullet$ is *associative*.

Haskell Definition

```haskell
class Semigroup a where
    (<>) :: a -> a -> a
```

1. **Associativity**: $(a \bullet (b \bullet c)) = ((a \bullet b) \bullet c)$

Example

```haskell
instance Semigroup [a] where
    (<>) = (++)
```

Monoid

A *monoid* is a semigroup $(S, \bullet)$ equipped with a special *identity element*.

Haskell Definition

```
class (Semigroup a) => Monoid a where
    mempty :: a
```

1. **Identity**: $(mempty \bullet x) = x = (x \bullet mempty)$

Example

```
instance Monoid [a] where
  mempty = []
```

Inductive Proofs

Suppose we want to prove that a property $P(n)$ holds for all natural numbers n. Remember that the set of natural numbers $\mathbb{N}$ can be defined as follows:

Definition of Natural Numbers

1. 0 is a natural number.
2. For any natural number n, $n + 1$ is also a natural number.

Therefore, to show $P(n)$ for all n, it suffices to show:

1. $P(0)$ (the *base case*), and
2. assuming $P(k)$ (the *inductive hypothesis*),
   $\Rightarrow P(k + 1)$ (the *inductive case*).

Natural Numbers Example

```haskell
data Nat = Zero
         | Succ Nat

add :: Nat -> Nat -> Nat
add Zero     n = n
add (Succ a) b = add a (Succ b)

one = Succ Zero
two = Succ (Succ Zero)
```

Example (1 + 1 = 2)

Prove `` one `add` one = two `` (done in editor)

Induction on Lists

Haskell lists can be defined similarly to natural numbers.

Definition of Haskell Lists

1. `[]` is a list.
2. For any list `xs`, `x:xs` is also a list (for any item `x`).

This means, if we want to prove that a property $P(\texttt{ls})$ holds for all lists `ls`, it suffices to show:

1. $P(\texttt{[]})$ (the base case)
2. $P(\texttt{x:xs})$ for all items x, assuming the inductive hypothesis $P(\texttt{xs})$.

List Monoid Example

```haskell
(++) :: [a] -> [a] -> [a]
(++) []     ys = ys              -- 1
(++) (x:xs) ys = x : xs ++ ys   -- 2
```

Example (Monoid)

Prove for all `xs, ys, zs`: `((xs ++ ys) ++ zs) = (xs ++ (ys ++ zs))`

Additionally Prove

1. for all `xs`: `[] ++ xs == xs`
2. for all `xs`: `xs ++ [] == xs`

(done in editor)

List Reverse Example

```
(++) :: [a] -> [a] -> [a]
(++) []     ys = ys                    -- 1
(++) (x:xs) ys = x : xs ++ ys          -- 2

reverse :: [a] -> [a]
reverse []     = []                    -- A
reverse (x:xs) = reverse xs ++ [x]     -- B
```

Example

To Prove for all `ls`: `reverse (reverse ls) == ls`
(done in editor)
First Prove for all `ys`: `reverse (ys ++ [x]) = x:reverse ys`
(done in editor)

Graphics and Artwork

```
data PictureObject
    = Path    [Point]       Colour LineStyle
    | Circle  Point Float   Colour LineStyle FillStyle
    | Polygon [Point]       Colour LineStyle FillStyle
    | Ellipse Point Float Float Float
              Colour LineStyle FillStyle
    deriving (Show, Eq)

type Picture = [PictureObject]
```

Homework

1. Last week's quiz is due before on Friday. Make sure you submit your answers.
2. Do the first programming exercise, and ask us on Discourse if you get stuck. It is due by the start if my next lecture (in 7 days).
3. This week's quiz is also up, it's due next Friday (in 7 days).

COMP3141

Software System Design and Implementation

Induction, Data Types and Type Classes

Dr. Christine Rizkallah
UNSW Sydney
Term 2 2021

Recap: Induction

Suppose we want to prove that a property $P(n)$ holds for all natural numbers n.
Remember that the set of natural numbers $\mathbb{N}$ can be defined as follows:

Definition of Natural Numbers

1. 0 is a natural number.
2. For any natural number n, $n + 1$ is also a natural number.

Recap: Induction

Therefore, to show $P(n)$ for all n, it suffices to show:

1. $P(0)$ (the *base case*), and
2. assuming $P(k)$ (the *inductive hypothesis*),
$\Rightarrow P(k+1)$ (the *inductive case*).

Example

Show that $f(n) = n^2$ for all $n \in \mathbb{N}$, where:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2n - 1 + f(n-1) & \text{if } n > 0 \end{cases}$$

(done on iPad)

Induction on Lists

Haskell lists can be defined similarly to natural numbers.

Definition of Haskell Lists

1. `[]` is a list.
2. For any list `xs`, `x:xs` is also a list (for any item `x`).

Induction on Lists

Haskell lists can be defined similarly to natural numbers.

Definition of Haskell Lists

1. `[]` is a list.
2. For any list `xs`, `x:xs` is also a list (for any item `x`).

This means, if we want to prove that a property $P(\texttt{ls})$ holds for all lists `ls`, it suffices to show:

1. $P(\texttt{[]})$ (the base case)
2. $P(\texttt{x:xs})$ for all items x, assuming the inductive hypothesis $P(\texttt{xs})$.

Induction on Lists: Example

```haskell
sum :: [Int] -> Int
sum []     = 0            -- 1
sum (x:xs) = x + sum xs -- 2

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

Example

Prove for all `ls`:

`sum ls == foldr (+) 0 ls`

(done on iPad)

Custom Data Types

So far, we have seen type synonyms using the `type` keyword. For a graphics library, we might define:

```
type Point   = (Float, Float)
type Vector  = (Float, Float)
type Line    = (Point, Point)
type Colour  = (Int, Int, Int, Int) -- RGBA

movePoint :: Point -> Vector -> Point
movePoint (x,y) (dx,dy) = (x + dx, y + dy)
```

But these definitions allow `Point`s and `Vector`s to be used interchangeably, increasing the likelihood of errors.

Product Types

We can define our own compound types using the `data` keyword:

Type name | Constructor name | Constructor argument types

```haskell
data Point = Point Float Float
           deriving (Show, Eq)

data Vector = Vector Float Float
            deriving (Show, Eq)

movePoint :: Point -> Vector -> Point
movePoint (Point x y) (Vector dx dy)
   = Point (x + dx) (y + dy)
```

Records

We could define Colour similarly:

```
data Colour = Colour Int Int Int Int
```

But this has so many parameters, it's hard to tell which is which.

Records

We could define Colour similarly:

```
data Colour = Colour Int Int Int Int
```

But this has so many parameters, it's hard to tell which is which.
Haskell lets us declare these types as *records*, which is identical to the declaration style on the previous slide, but also gives us projection functions and record syntax:

```
data Colour = Colour { redC     :: Int
                     , greenC   :: Int
                     , blueC    :: Int
                     , opacityC :: Int
                     } deriving (Show, Eq)
```

Here, the code `redC (Colour 255 128 0 255)` gives 255.

Enumeration Types

Similar to enums in C and Java, we can define types to have one of a set of predefined values:

```
data LineStyle = Solid
               | Dashed
               | Dotted
               deriving (Show, Eq)

data FillStyle = SolidFill | NoFill
               deriving (Show, Eq)
```

Types with more than one constructor are called *sum types*.

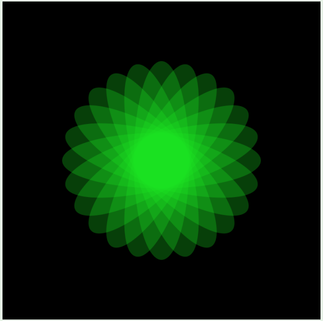
Algebraic Data Types

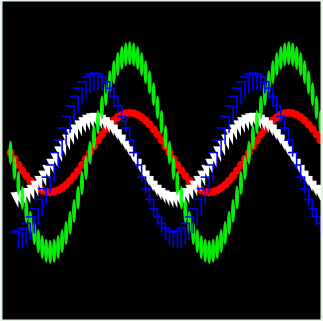
Just as the `Point` constructor took two `Float` arguments, constructors for sum types can take parameters too, allowing us to model different kinds of shape:

```
data PictureObject
    = Path    [Point]       Colour LineStyle
    | Circle  Point Float   Colour LineStyle FillStyle
    | Polygon [Point]       Colour LineStyle FillStyle
    | Ellipse Point Float Float Float
              Colour LineStyle FillStyle
    deriving (Show, Eq)

type Picture = [PictureObject]
```

Live Coding: Cool Graphics

Example (Ellipses and Curves)





Recursive and Parametric Types

Data types can also be defined with parameters, such as the well known `Maybe` type, defined in the standard library:

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

Recursive and Parametric Types

Data types can also be defined with parameters, such as the well known `Maybe` type, defined in the standard library:

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

Types can also be recursive. If lists weren't already defined in the standard library, we could define them ourselves:

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

Recursive and Parametric Types

Data types can also be defined with parameters, such as the well known `Maybe` type, defined in the standard library:

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

Types can also be recursive. If lists weren't already defined in the standard library, we could define them ourselves:

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

We can even define natural numbers, where 2 is encoded as `Succ(Succ Zero)`:

```
data Natural = Zero | Succ Natural
```

Types in Design

Sage Advice

An old adage due to Yaron Minsky (of Jane Street) is:

Make illegal states unrepresentable.

Choose types that *constrain* your implementation as much as possible. Then failure scenarios are eliminated automatically.

Types in Design

Sage Advice

An old adage due to Yaron Minsky (of Jane Street) is:

Make illegal states unrepresentable.

Choose types that *constrain* your implementation as much as possible. Then failure scenarios are eliminated automatically.

Example (Contact Details)

```
data Contact = C Name (Maybe Address) (Maybe Email)
```

is changed to:

```
data ContactDetails = EmailOnly Email | PostOnly Address
                    | Both Address Email
data Contact = C Name ContactDetails
```

What failure state is eliminated here?

Partial Functions

Failure to follow Yaron's excellent advice leads to partial functions.

Definition

A *partial function* is a function not defined for all possible inputs.
Examples: `head`, `tail`, `(!!)`, division

Partial functions are to be avoided, because they cause your program to crash if undefined cases are encountered.

Partial Functions

Failure to follow Yaron's excellent advice leads to partial functions.

Definition

A *partial function* is a function not defined for all possible inputs.

Examples: `head`, `tail`, `(!!)`, division

Partial functions are to be avoided, because they cause your program to crash if undefined cases are encountered.

To eliminate partiality, we must either:

- enlarge the codomain, usually with a `Maybe` type:

```haskell
safeHead :: [a] -> Maybe a -- Q: How is this safer?
safeHead (x:xs) = Just x
safeHead []     = Nothing
```

Partial Functions

Failure to follow Yaron's excellent advice leads to partial functions.

Definition

A *partial function* is a function not defined for all possible inputs.

Examples: `head`, `tail`, `(!!)`, division

Partial functions are to be avoided, because they cause your program to crash if undefined cases are encountered.

To eliminate partiality, we must either:

- enlarge the codomain, usually with a `Maybe` type:

```
safeHead :: [a] -> Maybe a -- Q: How is this safer?
safeHead (x:xs) = Just x
safeHead []     = Nothing
```

- Or we must constrain the domain to be more specific:

```
safeHead' :: NonEmpty a -> a -- Q: How to define?
```

Type Classes

You have already seen functions such as:

- `compare`
- `(==)`
- `(+)`
- `show`

that work on multiple types, and their corresponding constraints on type variables `Ord`, `Eq`, `Num` and `Show`.

Type Classes

You have already seen functions such as:

- `compare`
- `(==)`
- `(+)`
- `show`

that work on multiple types, and their corresponding constraints on type variables `Ord`, `Eq`, `Num` and `Show`.

These constraints are called *type classes*, and can be thought of as a set of types for which certain operations are implemented.

Show

The `Show` type class is a set of types that can be converted to strings. It is defined like:

```haskell
class Show a where  -- nothing to do with OOP
  show :: a -> String
```

Show

The `Show` type class is a set of types that can be converted to strings. It is defined like:

```
class Show a where  -- nothing to do with OOP
  show :: a -> String
```

Types are added to the type class as an *instance* like so:

```
instance Show Bool where
  show True  = "True"
  show False = "False"
```

Show

The `Show` type class is a set of types that can be converted to strings. It is defined like:

```
class Show a where  -- nothing to do with OOP
  show :: a -> String
```

Types are added to the type class as an *instance* like so:

```
instance Show Bool where
  show True  = "True"
  show False = "False"
```

We can also define instances that depend on other instances:

```
instance Show a => Show (Maybe a) where
  show (Just x) = "Just " ++ show x
  show Nothing  = "Nothing"
```

Fortunately for us, Haskell supports automatically `deriving` instances for some classes, including `Show`.

Read

Type classes can also overload based on the type returned, unlike similar features like Java's interfaces:

```
class Read a where
  read :: String -> a
```

Some examples:

- `read "34" :: Int`

Read

Type classes can also overload based on the type returned, unlike similar features like Java's interfaces:

```
class Read a where
  read :: String -> a
```

Some examples:

- `read "34" :: Int`
- `read "22" :: Char`

Read

Type classes can also overload based on the type returned, unlike similar features like Java's interfaces:

```
class Read a where
  read :: String -> a
```

Some examples:

- `read "34" :: Int`
- `read "22" :: Char` Runtime error!
- `show (read "34") :: String`

Read

Type classes can also overload based on the type returned, unlike similar features like Java's interfaces:

```
class Read a where
  read :: String -> a
```

Some examples:

- `read "34" :: Int`
- `read "22" :: Char` Runtime error!
- `show (read "34") :: String` Type error!

Semigroup

Semigroups

A *semigroup* is a pair of a set S and an operation $\bullet : S \to S \to S$ where the operation $\bullet$ is *associative*.

Semigroup

Semigroups

A *semigroup* is a pair of a set S and an operation $\bullet : S \rightarrow S \rightarrow S$ where the operation $\bullet$ is *associative*.

Associativity is defined as, for all a, b, c:

$$(a \bullet (b \bullet c)) = ((a \bullet b) \bullet c)$$

Haskell has a type class for semigroups! The associativity law is enforced only by programmer discipline:

```
class Semigroup s where
  (<>) :: s -> s -> s
  -- Law: (<>) must be associative.
```

What instances can you think of?

Semigroup

Lets implement additive colour mixing:

```
instance Semigroup Colour where
  Colour r1 g1 b1 a1 <> Colour r2 g2 b2 a2
      = Colour (mix r1 r2)
               (mix g1 g2)
               (mix b1 b2)
               (mix a1 a2)
    where
      mix x1 x2 = min 255 (x1 + x2)
```

Observe that associativity is satisfied.

Monoid

Monoids

A *monoid* is a semigroup $(S, \bullet)$ equipped with a special *identity element* $z : S$ such that $x \bullet z = x$ and $z \bullet y = y$ for all x, y.

Monoid

Monoids

A *monoid* is a semigroup $(S, \bullet)$ equipped with a special *identity element* $z : S$ such that $x \bullet z = x$ and $z \bullet y = y$ for all x, y.

```
class (Semigroup a) => Monoid a where
  mempty :: a
```

Monoid

Monoids

A *monoid* is a semigroup $(S, \bullet)$ equipped with a special *identity element* $z : S$ such that $x \bullet z = x$ and $z \bullet y = y$ for all x, y.

```
class (Semigroup a) => Monoid a where
  mempty :: a
```

For colours, the identity element is transparent black:

```
instance Monoid Colour where
  mempty = Colour 0 0 0 0
```

For each of the semigroups discussed previously:

- Are they monoids?
- If so, what is the identity element?

Monoid

Monoids

A *monoid* is a semigroup $(S, \bullet)$ equipped with a special *identity element* $z : S$ such that $x \bullet z = x$ and $z \bullet y = y$ for all x, y.

```haskell
class (Semigroup a) => Monoid a where
  mempty :: a
```

For colours, the identity element is transparent black:

```haskell
instance Monoid Colour where
  mempty = Colour 0 0 0 0
```

For each of the semigroups discussed previously:

- Are they monoids?
- If so, what is the identity element?

Are there any semigroups that are not monoids?

Monoid

Monoids

A *monoid* is a semigroup $(S, \bullet)$ equipped with a special *identity element* $z : S$ such that $x \bullet z = x$ and $z \bullet y = y$ for all x, y.

```haskell
class (Semigroup a) => Monoid a where
  mempty :: a
```

For colours, the identity element is transparent black:

```haskell
instance Monoid Colour where
  mempty = Colour 0 0 0 0
```

For each of the semigroups discussed previously:

- Are they monoids?
- If so, what is the identity element?

Are there any semigroups that are not monoids?

Non-empty lists, maximum

Newtypes

There are multiple possible monoid instances for numeric types like `Integer`:

- The operation `(+)` is associative, with identity element `0`
- The operation `(*)` is associative, with identity element `1`

Newtypes

There are multiple possible monoid instances for numeric types like `Integer`:

- The operation `(+)` is associative, with identity element `0`
- The operation `(*)` is associative, with identity element `1`

Haskell doesn't use any of these, because there can be only one instance per type per class in the entire program (including all dependencies and libraries used).

A common technique is to define a separate type that is represented identically to the original type, but can have its own, different type class instances.

Newtypes

There are multiple possible monoid instances for numeric types like `Integer`:

- The operation `(+)` is associative, with identity element `0`
- The operation `(*)` is associative, with identity element `1`

Haskell doesn't use any of these, because there can be only one instance per type per class in the entire program (including all dependencies and libraries used).

A common technique is to define a separate type that is represented identically to the original type, but can have its own, different type class instances.

In Haskell, this is done with the `newtype` keyword.

Newtypes

A `newtype` declaration is much like a `data` declaration except that there can be only one constructor and it must take exactly one argument:

```
newtype Score = S Integer

instance Semigroup Score where
  S x <> S y = S (x + y)

instance Monoid Score where
  mempty = S 0
```

Here, `Score` is represented identically to `Integer`, and thus no performance penalty is incurred to convert between them.

In general, newtypes are a great way to prevent mistakes. Use them frequently!

Ord

`Ord` is a type class for inequality comparison:

```
class Ord a where
  (<=) :: a -> a -> Bool
```

What laws should instances satisfy?

Ord

`Ord` is a type class for inequality comparison:

```
class Ord a where
  (<=) :: a -> a -> Bool
```

What laws should instances satisfy?

For all `x`, `y`, and `z`:

1. *Reflexivity*: `x <= x`.

Ord

`Ord` is a type class for inequality comparison:

```
class Ord a where
  (<=) :: a -> a -> Bool
```

What laws should instances satisfy?

For all `x`, `y`, and `z`:

1. *Reflexivity*: `x <= x`.
2. *Transitivity*: If `x <= y` and `y <= z` then `x <= z`.

Ord

`Ord` is a type class for inequality comparison:

```
class Ord a where
  (<=) :: a -> a -> Bool
```

What laws should instances satisfy?

For all `x`, `y`, and `z`:

1. *Reflexivity*: `x <= x`.
2. *Transitivity*: If `x <= y` and `y <= z` then `x <= z`.
3. *Antisymmetry*: If `x <= y` and `y <= x` then `x == y`.

Ord

`Ord` is a type class for inequality comparison:

```
class Ord a where
  (<=) :: a -> a -> Bool
```

What laws should instances satisfy?

For all `x`, `y`, and `z`:

1. *Reflexivity*: `x <= x`.
2. *Transitivity*: If `x <= y` and `y <= z` then `x <= z`.
3. *Antisymmetry*: If `x <= y` and `y <= x` then `x == y`.
4. *Totality*: Either `x <= y` or `y <= x`

Ord

`Ord` is a type class for inequality comparison:

```
class Ord a where
  (<=) :: a -> a -> Bool
```

What laws should instances satisfy?

For all `x`, `y`, and `z`:

1. *Reflexivity*: `x <= x`.
2. *Transitivity*: If `x <= y` and `y <= z` then `x <= z`.
3. *Antisymmetry*: If `x <= y` and `y <= x` then `x == y`.
4. *Totality*: Either `x <= y` or `y <= x`

Relations that satisfy these four properties are called *total orders*. Without the fourth (totality), they are called *partial orders*.

Eq

Eq is a type class for equality or equivalence:

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

What laws should instances satisfy?

Eq

`Eq` is a type class for equality or equivalence:

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

What laws should instances satisfy?

For all `x`, `y`, and `z`:

1. *Reflexivity*: `x == x`.

Eq

`Eq` is a type class for equality or equivalence:

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

What laws should instances satisfy?

For all `x`, `y`, and `z`:

1. *Reflexivity*: `x == x`.
2. *Transitivity*: If `x == y` and `y == z` then `x == z`.

Eq

Eq is a type class for equality or equivalence:

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

What laws should instances satisfy?

For all `x`, `y`, and `z`:

1. *Reflexivity*: `x == x`.
2. *Transitivity*: If `x == y` and `y == z` then `x == z`.
3. *Symmetry*: If `x == y` then `y == x`.

Eq

`Eq` is a type class for equality or equivalence:

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

What laws should instances satisfy?

For all `x`, `y`, and `z`:

1. *Reflexivity*: `x == x`.
2. *Transitivity*: If `x == y` and `y == z` then `x == z`.
3. *Symmetry*: If `x == y` then `y == x`.

Relations that satisfy these are called *equivalence relations*.

Eq

Eq is a type class for equality or equivalence:

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

What laws should instances satisfy?

For all x, y, and z:

1. *Reflexivity*: `x == x`.
2. *Transitivity*: If `x == y` and `y == z` then `x == z`.
3. *Symmetry*: If `x == y` then `y == x`.

Relations that satisfy these are called *equivalence relations*.

Some argue that the Eq class should be only for *equality*, requiring stricter laws like:

If `x == y` then `f x == f y` for all functions `f`

But this is debated.

Types and Values

Haskell is actually comprised of two languages.

- The *value-level* language, consisting of expressions such as `if`, `let`, `3` etc.

Types and Values

Haskell is actually comprised of two languages.

- The *value-level* language, consisting of expressions such as `if`, `let`, `3` etc.
- The *type-level* language, consisting of types `Int`, `Bool`, synonyms like `String`, and type *constructors* like `Maybe`, `(->)`, `[ ]` etc.

Types and Values

Haskell is actually comprised of two languages.

- The *value-level* language, consisting of expressions such as `if`, `let`, `3` etc.
- The *type-level* language, consisting of types `Int`, `Bool`, synonyms like `String`, and type *constructors* like `Maybe`, `(->)`, `[ ]` etc.

This type level language itself has a type system!

Kinds

Just as terms in the value level language are given types, terms in the type level language are given *kinds*.

The most basic kind is written as `*`.

- Types such as `Int` and `Bool` have kind `*`.
- Seeing as `Maybe` is parameterised by one argument, `Maybe` has kind `* -> *`: given a type (e.g. `Int`), it will return a type (`Maybe Int`).

Lists

Suppose we have a function:

```
toString :: Int -> String
```

And we also have a function to give us some numbers:

```
getNumbers :: Seed -> [Int]
```

How can I compose `toString` with `getNumbers` to get a function `f` of type `Seed -> [String]`?

Lists

Suppose we have a function:

```
toString :: Int -> String
```

And we also have a function to give us some numbers:

```
getNumbers :: Seed -> [Int]
```

How can I compose `toString` with `getNumbers` to get a function `f` of type `Seed -> [String]`?

Answer: we use `map`:

```
f = map toString . getNumbers
```

Maybe

Suppose we have a function:

```haskell
toString :: Int -> String
```

And we also have a function that may give us a number:

```haskell
tryNumber :: Seed -> Maybe Int
```

How can I compose `toString` with `tryNumber` to get a function `f` of type `Seed -> Maybe String`?

Maybe

Suppose we have a function:

```
toString :: Int -> String
```

And we also have a function that may give us a number:

```
tryNumber :: Seed -> Maybe Int
```

How can I compose `toString` with `tryNumber` to get a function `f` of type `Seed -> Maybe String`?

We want a `map` function but for the `Maybe` type:

```
f = maybeMap toString . tryNumber
```

Let's implement it.

Functor

All of these functions are in the interface of a single type class, called `Functor`.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Functor

All of these functions are in the interface of a single type class, called `Functor`.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Unlike previous type classes we've seen like `Ord` and `Semigroup`, `Functor` is over types of kind `* -> *`.

Functor

All of these functions are in the interface of a single type class, called `Functor`.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Unlike previous type classes we've seen like `Ord` and `Semigroup`, `Functor` is over types of kind `* -> *`.

Instances for:

- Lists
- `Maybe`
- Tuples (how?)
- Functions (how?)

Demonstrate in live-coding

Functor Laws

The functor type class must obey two laws:

Functor Laws

1. `fmap id == id`
2. `fmap f . fmap g == fmap (f . g)`

Functor Laws

The functor type class must obey two laws:

Functor Laws

1. `fmap id == id`
2. `fmap f . fmap g == fmap (f . g)`

In Haskell's type system it's impossible to make a total `fmap` function that satisfies the first law but violates the second.

This is due to *parametricity*, a property we will return to in Week 8 or 9

Homework

1. Do the first programming exercise, and ask us on Piazza if you get stuck. It will be due in exactly 1 week from the start of the Friday lecture.
2. Last week's quiz is due this Friday. Make sure you submit your answers.
3. This week's quiz is also up, due next Friday (the Friday after this one).

COMP3141

Software System Design and Implementation

Property Based Testing Practice

Curtis Millar
CSE, UNSW
18 June 2021

Revision Lecture

- There will be a revision lecture in weeks 4, 7, & 9
- Current plan is for Thursday at 3pm
- Reply to post on forum if this is an issue
- Vote for revision topics in the *Revision Lectures* category on the course forum
- If you want a topic discussed and it is not listed there, post there with the topic or question you want discussed.

Exercise 1

1. **Simple Picture**: add the chimney and smoke
2. **Moving Objects**: implement `movePictureObject`
3. **Generating a Picture**: generate pictures of circles using `simpleCirclePic`

Property Based Testing

Key idea: Generate random input values, and test properties by running them.

Example (QuickCheck Property)

```
prop_reverseApp xs ys =
    reverse (xs ++ ys) == reverse ys ++ reverse xs
```

Haskell's *QuickCheck* is the first library ever invented for property-based testing. The concept has since been ported to Erlang, Scheme, Common Lisp, Perl, Python, Ruby, Java, Scala, F#, OCaml, Standard ML, C and C++.

Mersenne Prime Example

Example (Demo Task)

- The n^{th} Mersenne number $M_n = 2^{n-1}$.
- M_2, M_3, M_5 and M_7 are all prime numbers.
- **Conjecture:** $\forall\, n.prime(n) \implies prime(2^{n-1})$

Let's try using QuickCheck to answer this question.

After a small number of guesses and fractions of a second, QuickCheck found a counter-example to this conjecture: 11.

It took humanity about two thousand years to do the same.

Semigroup and Monoid Properties

Last week we proved by hand that a list forms a semigroup with ++ as its associative operator and a monoid with [] as its identity element.
We can show the same properties much faster (although less completely) with property based testing.

QuickCheck Properties

```haskell
-- Semigroup laws
prop_listAssociative xs ys zs = ((xs ++ ys) ++ zs) == (xs ++ (ys ++ zs))

-- Monoid laws
prop_listLeftIdentity xs = xs == [] ++ xs
prop_listRightIdentity xs = xs == xs ++ []
```

Reverse Involution

Last week we also proved by hand that the `reverse` function is an *involution*.
This took over twenty minutes.
Let's see how long it takes QuickCheck.

QuickCheck Property

```
prop_reverseInvolution xs = reverse (reverse xs) == xs
```

Ransom Note Example

Example (Demo Task)

Given a magazine (in `String` form), is it possible to create a ransom message (in `String` form) from characters in the magazine.

```haskell
canMakeRansom :: RansomNote -> Magazine -> Bool
```

1. Write a specification
2. Create an efficient implementation
3. Test the implementation

In Haskell.

Proofs

Proofs:

- Proofs must make some assumptions about the environment and the semantics of the software.
- Proof complexity grows with implementation complexity, sometimes drastically.
- If software is incorrect, a proof attempt might simply become stuck: we do not always get constructive negative feedback.
- Proofs can be labour and time intensive ($$$), or require highly specialised knowledge ($$$).

Testing

Compared to proofs:

- Tests typically run the actual program, so requires fewer assumptions about the language semantics or operating environment.
- Test complexity does not grow with implementation complexity, so long as the specification is unchanged.
- Incorrect software when tested leads to immediate, debuggable counterexamples.
- Testing is typically cheaper and faster than proving.
- Tests care about efficiency and computability, unlike proofs.

We lose some assurance, but gain some convenience ($$$).

Verification versus Validation

"Testing shows the presence, but not the absence of bugs."
– Dijkstra (1969)

Testing is essential but is insufficient for safety-critical applications.

Homework

1. Last week's quiz is due on Friday at 6pm. Make sure you submit your answers.
2. The second programming exercise is due by the start if my next lecture (in 7 days).
3. This week's quiz is also up, it's due next Friday (in 7 days).
4. Post and vote for revision topics in the *Revision Lectures* category on the forum.

COMP3141

Software System Design and Implementation

Property Based Testing; Lazy Evaluation

Dr. Christine Rizkallah
UNSW Sydney
Term 2 2021

Free Properties

Haskell already ensures certain properties automatically with its language design and type system.

1. Memory is accessed where and when it is safe and permitted to be accessed (*memory safety*).
2. Values of a certain static type will actually have that type at run time.
3. Programs that are well-typed will not lead to undefined behaviour (*type safety*).
4. All functions are *pure*: Programs won't have side effects not declared in the type. (*purely functional programming*)

$\Rightarrow$ Most of our properties focus on the *logic of our program*.

Logical Properties

We have already seen a few examples of logical properties.

Example (Properties)

1. `reverse` is an *involution*: `reverse (reverse xs) == xs`
2. right identity for `(++)`: `xs ++ [] == xs`
3. transitivity of `(>)`: `(a > b)` $\wedge$ `(b > c)` $\Rightarrow$ `(a > c)`

The set of properties that capture all of our requirements for our program is called the *functional correctness specification* of our software.

This defines what it means for software to be correct.

Proofs

Last week we saw some *proof methods* for Haskell programs. We could prove that our implementation meets its functional correctness specification.

Such proofs certainly offer a high degree of assurance, but:

- Proofs must make some assumptions about the environment and the semantics of the software.
- Proof complexity grows with implementation complexity, sometimes drastically.
- If software is incorrect, a proof attempt might simply become stuck: we do not always get constructive negative feedback.
- Proofs can be labour and time intensive ($$$), or require highly specialised knowledge ($$$).

Testing

Compared to proofs:

- Tests typically run the actual program, so requires fewer assumptions about the language semantics or operating environment.
- Test complexity does not grow with implementation complexity, so long as the specification is unchanged.
- Incorrect software when tested leads to immediate, debuggable counterexamples.
- Testing is typically cheaper and faster than proving.
- Tests care about efficiency and computability, unlike proofs.

We lose some assurance, but gain some convenience ($$$).

Property Based Testing

Key idea: Generate random input values, and test properties by running them.

Example (QuickCheck Property)

```
prop_reverseApp xs ys =
    reverse (xs ++ ys) == reverse ys ++ reverse xs
```

Haskell's *QuickCheck* is the first library ever invented for property-based testing. The concept has since been ported to Erlang, Scheme, Common Lisp, Perl, Python, Ruby, Java, Scala, F#, OCaml, Standard ML, C and C++.

PBT vs. Unit Testing

- Properties are more compact than unit tests, and describe more cases.
  $\Rightarrow$ Less testing code
- Property-based testing heavily depends on test data generation:
  - Random inputs may not be as informative as hand-crafted inputs
    $\Rightarrow$ use shrinking
  - Random inputs may not cover all necessary corner cases:
    $\Rightarrow$ use a coverage checker
  - Random inputs must be generated for user-defined types:
    $\Rightarrow$ QuickCheck includes functions to build custom generators
- By increasing the number of random inputs, we improve code coverage in PBT.

Test Data Generation

Data which can be generated randomly is represented by the following type class:

```
class Arbitrary a where
  arbitrary :: Gen a  -- more on this later
  shrink :: a -> [a]
```

Most of the types we have seen so far implement `Arbitrary`.

Shrinking

The `shrink` function is for when test cases fail. If a given input `x` fails, QuickCheck will try all inputs in `shrink x`; repeating the process until the smallest possible input is found.

Testable Types

The type of the `quickCheck` function is:

```
                                        -- more on IO later
quickCheck :: (Testable a) => a -> IO ()
```

The `Testable` type class is the class of things that can be converted into properties. This includes:

- `Bool` values
- QuickCheck's built-in `Property` type
- Any function from an `Arbitrary` input to a `Testable` output:

```
instance (Arbitrary i, Testable o)
      => Testable (i -> o) ...
```

Thus the type `[Int] -> [Int] -> Bool` (as used earlier) is `Testable`.

Simple example

Is this function reflexive?

```
divisible :: Integer -> Integer -> Bool
divisible x y = x `mod` y == 0

prop_refl :: Integer -> Bool
prop_refl x = divisible x x
```

- Encode pre-conditions with the (==>) operator:

  ```
  prop_refl :: Integer -> Property
  prop_refl x = x > 0 ==> divisible x x
  ```

  (but may generate a lot of spurious cases)
- or select different generators with modifier `newtypes`.

  ```
  prop_refl :: Positive Integer -> Bool
  prop_refl (Positive x) = divisible x x
  ```

  (but may require you to define custom generators)

Words and Inverses

Example (Inverses)

```
words   :: String -> [String]
unwords :: [String] -> String
```

We might expect `unwords` to be the inverse of `words` and vice versa. Let's find out!

Lessons: Properties aren't always what you expect!

Merge Sort

Example (Merge Sort)

Recall merge sort, the sorting algorithm that is reliably $\mathcal{O}(n \log n)$ time complexity.

- If the list is empty or one element, return that list.
- Otherwise, we:
  1. Split the input list into two sublists.
  2. Recursively sort the two sublists.
  3. Merge the two sorted sublists into one sorted list in linear time.

Applying our bottom up design, let's posit:

```
split :: [a] -> ([a],[a])
merge :: (Ord a) => [a] -> [a] -> [a]
```

Split

```
split :: [a] -> ([a],[a])
```

What is a good specification of split?

- Each element of the input list occurs in one of the two output lists, the same number of times.
- The two output lists consist only of elements from the input list.

Because of its usefulness later, we'll define this in terms of a permutation predicate.

Merge

```haskell
merge :: (Ord a) => [a] -> [a] -> [a]
```

What is a good specification of merge?

- Each element of the output list occurs in one of the two input lists, the same number of times.
- The two input lists consist solely of elements from the output list.
- **Important:** If the input lists are sorted, then the output list is sorted.

Overall

```
mergesort :: (Ord a) => [a] -> [a]
```

What is a good specification of mergesort?

- The output list is sorted.
- The output list is a permutation of the input list.

We can prove this as a consequence of the previous specifications which we tested. We can also just write integration properties that test the composition of these functions together.

Redundant Properties

Some properties are technically redundant (i.e. implied by other properties in the specification), but there is some value in testing them anyway:

- They may be more efficient than full functional correctness tests, consuming less computing resources to test.
- They may be more fine-grained to give better test coverage than random inputs for full functional correctness tests.
- They provide a good sanity check to the full functional correctness properties.
- Sometimes full functional correctness is not easily computable but tests of weaker properties are.

These redundant properties include unit tests. We can (and should) combine both approaches!

What are some redundant properties of `mergesort`?

Test Quality

How good are your tests?

- Have you checked that every special case works correctly?
- Is all code exercised in the tests?
- Even if all code is exercised, is it exercised in all contexts?

Coverage checkers are useful tools to partially quantify this.

Types of Coverage

Branch/Decision Coverage
All conditional branches executed?

Entry/Exit Coverage
All function calls executed?

Function Coverage
All functions executed?

Statement/Expression Coverage
All expressions executed?

Path Coverage
All behaviours executed?
very hard!

Haskell Program Coverage

Haskell Program Coverage (or `hpc`) is a GHC-bundled tool to measure function, branch and expression coverage.

Sum to n

```
sumTo :: Integer -> Integer
sumTo 0 = 0
sumTo n = sumTo (n-1) + n
```

This crashes when given a large number. Why?

Sum to n, redux

```
sumTo' :: Integer -> Integer -> Integer
sumTo' a 0 = a
sumTo' a n = sumTo' (a+n) (n-1)

sumTo = sumTo' 0
```

This still crashes when given a large number. Why?

This is called a space leak, and is one of the main drawbacks of Haskell's lazy evaluation method.

Lazy Evaluation

Haskell is lazily evaluated, also called call-by-need.
This means that expressions are only evaluated when they are needed to compute a result for the user.
We can force the previous program to evaluate its accumulator by using a bang pattern, or the primitive operation seq:

```haskell
sumTo' :: Integer -> Integer -> Integer
sumTo' !a 0 = a
sumTo' !a n = sumTo' (a+n) (n-1)
```

```haskell
sumTo' :: Integer -> Integer -> Integer
sumTo' a 0 = a
sumTo' a n = let a' = a + n in a' `seq` sumTo' a' (n-1)
```

Advantages

Lazy Evaluation has many advantages:

- It enables equational reasoning even in the presence of partial functions and non-termination.
- It allows functions to be decomposed without sacrificing efficiency, for example: `minimum = head . sort` is, depending on sorting algorithm, possibly $\mathcal{O}(n)$. John Hughes demonstrates $\alpha\beta$ pruning from AI as a larger example.[1]
- It allows for circular programming and infinite data structures, which allow us to express more things as pure functions.

Problem

In one pass over a list, replace every element of the list with its maximum.

[1] J. Hughes, "Why Functional Programming Matters", Comp. J., 1989

Infinite Data Structures

Laziness lets us define data structures that extend infinitely. Lists are a common example, but it also applies to trees or any user-defined data type:

```
ones = 1 : ones
```

Many functions such as `take`, `drop`, `head`, `tail`, `filter` and `map` work fine on infinite lists!

```
naturals = 0 : map (1+) naturals
--or
naturals = map sum (inits ones)
```

How about fibonacci numbers?

```
fibs = 1:1:zipWith (+) fibs (tail fibs)
```

Homework

1. First programming exercise is due on Friday 12pm.
2. Second exercise is now out, due the following Friday 12pm.
3. Last week's quiz is due on Friday 6pm. Make sure you submit your answers.
4. This week's quiz is also up, due the following Friday 6pm.

COMP3141

Software System Design and Implementation

Data Invariants, Abstraction and Refinement Practice

Curtis Millar
CSE, UNSW
25 June 2021

Sort Properties

1. `sortFn xs == sortFn (reverse xs)`
2. ``x `elem` sortFn (xs ++ [x] ++ ys)``
3. `isSorted (sortFn xs)`
4. `length xs == length (sortFn xs)`
5. `sortFn xs == insertionSort xs`

Dodgy Sort

1. Satisfy only (2) and (4)
2. Satisfy only (1), (2), and (3)
3. Satisfy only (1), (3), and (4)
4. Satisfy only (1), (2), (3), and (4)

Fractal Art

- Let's take a look at the gallery
- Assess your peers
  1. Is the function which generates the image a recursive function?
  2. Is the picture function given parameters that influence at least two aspects of the image other than recursion depth, size, and colour?
  3. Is it a real attempt to generate a nice image?
- Online form to review peers art & implementation on course website soon.

Data Invariants

- Data invariants are statements that must always be true of a data structure. We generally represent these invariants as a *wellformedness predicate*, a function that tests whether a value is well-formed.
- Data invaraints must be shown to be true for all *constructors* of a data type. The output of any constructor must satisfy the wellformedness predicate.

```
constructor :: .. -> X
```

- Data invaraints must also be shown to be true for all functions that transform the value of a data type. The output of these functions must satisfy the wellformedness predicate only if the input does.

```
fn :: .. -> X -> X
```

Abstract Data Types

- ADTs allow us to encapsulate the implementation of a data type by restricting access to which functions can be used construct, query, or transform a value from *outside* the module in which it is defined.
- The ability to restrict access to certain implementation details is generally dependant on the language.
- If all the externally visible functions maintain the data invariants then no external code can construct a value that ever violates them.

Refinement

- A relation from an *implementation* to an *abstract model* or an *abstract specification*.
- If an implementation *refines* a model or specification, all of its behaviour is represented in the model.
  A refinement is the opposite of an abstraction, which removes detail in favor of generality.
- In this course, the model and implementaion will present an indistingushable interface with different implementation details; they will be *equivalent*.

Data Refinement

- One datatype *refines* another if all provable properties of the first are also provably true of the second.
- We can demonstrate an *equivalence relation* between two data types if we can show that their interfaces are functionally equivalent, i.e., the produce the equivalent or equal outputs given equivalent inputs inputs. This is a *data refinement*.
- We choose which data type will be the *abstract model* which is the *definition* or *specification*.
- The other data type then becomes our *implementation*, i.e. the data type that we will actually use in the final system.
- We must show that the implementation is a refinement of the model or specification.

Data Refinement

Refinement and Specifications

In general, all functional correctness specifications can be expressed as:

1. all data invariants are maintained, and
2. the implementation is a refinement of an abstract correctness model.

There is a limit to the amount of abstraction we can do before they become useless for testing (but not necessarily for proving).

Warning

While abstraction can simplify proofs, abstraction does not reduce the fundamental complexity of verification, which is provably hard.

Editor Example

Consider this ADT interface for a text editor:

```
data Editor
einit :: String -> Editor
stringOf :: Editor -> String
moveLeft :: Editor -> Editor
moveRight :: Editor -> Editor
insertChar :: Char -> Editor -> Editor
deleteChar :: Editor -> Editor
```

Data Invariant Properties

```
prop_einit_ok          s = wellformed (einitA s)
prop_moveLeft_ok     a = wellformed (moveLeftA a)
prop_moveRight_ok    a = wellformed (moveRightA a)
prop_insertChar_ok x a = wellformed (insertCharA x a)
prop_deleteChar_ok   a = wellformed (deleteCharA a)
```

Editor Example: Abstract Model

Our conceptual abstract model is a string and a cursor position:

```
einitA s = A s 0
stringOfA (A s _) = s
moveLeftA  (A t c) = A t (max 0 (c-1))
moveRightA (A t c) = A t (min (length t) (c+1))
insertCharA x (A t c) = let (t1, t2) = splitAt c t
                        in A (t1 ++ [x] ++ t2) (c+1)
deleteCharA (A t c) = let (t1, t2) = splitAt c t
                        in A (t1 ++ drop 1 t2) c
```

But do we need to keep track of all that information in our implementation? No!

Concrete Implementation

Our concrete version will just maintain two strings, the `left` part (in reverse) and the `right` part of the cursor:

```
einit s = C [] s
stringOf (C ls rs) = reverse ls ++ rs
moveLeft (C (l:ls) rs) = C ls (l:rs)
moveLeft c = c
moveRight (C ls (r:rs)) = C (r:ls) rs
moveRight c = c
insertChar x (C ls rs) = C (x:ls) rs
deleteChar (C ls (_:rs)) = C ls rs
deleteChar c = c
```

Refinement Functions

Abstraction function to express our refinement relation in a QC-friendly way: such a function:

```
toAbstract :: Concrete -> Abstract
toAbstract (C ls rs) = A (reverse ls ++ rs) (length ls)
```

Properties with Abstraction Functions

```
prop_init_r s =
  toAbstract (einit s) == einitA s
prop_stringOf_r c =
  stringOf c == stringOfA (toAbstract c)
prop_moveLeft_r c =
  toAbstract (moveLeft c) == moveLeftA (toAbstract c)
prop_moveRight_r c =
  toAbstract (moveRight c) == moveRightA (toAbstract c)
prop_insertChar_r x c =
  toAbstract (insertChar x c)
  == insertCharA x (toAbstract c)
prop_deleteChar_r c =
  toAbstract (deleteChar c) == deleteCharA (toAbstract c)
```

Homework

1. Last week's quiz is due at 6pm. Make sure you submit your answers.
2. The third programming exercise is due by the start if my next lecture (in 7 days).
3. The first assignment is due by the start if my next lecture (in 7 days).
4. This week's quiz is also up, it's due next Friday (in 7 days).

COMP3141

Software System Design and Implementation

Data Invariants, Abstraction and Refinement

Christine Rizkallah
UNSW Sydney
Term 2 2021

Motivation

We've already seen how to prove and test correctness properties of our programs.

How do we come up with correctness properties in the first place?

Structure of a Module

A Haskell program will usually be made up of many modules, each of which exports one or more *data types*.

Typically a module for a data type $\mathtt{X}$ will also provide a set of functions, called *operations*, on $\mathtt{X}$.

- to construct the data type: $c :: \cdots \rightarrow \mathtt{X}$
- to query information from the data type: $q :: \mathtt{X} \rightarrow \cdots$
- to update the data type: $u :: \cdots \mathtt{X} \rightarrow \mathtt{X}$

A lot of software can be designed with this structure.

Example (Data Types)

A dictionary data type, with empty, insert and lookup.

Data Invariants

One source of properties is *data invariants*.

Data Invariants

Data invariants are properties that pertain to a particular data type.
Whenever we use operations on that data type, we want to know that our data invariants are maintained.

Example

- That a list of words in a dictionary is always in sorted order
- That a binary tree satisfies the search tree properties.
- That a date value will never be invalid (e.g. 31/13/2019).

Properties for Data Invariants

For a given data type `X`, we define a *wellformedness predicate*

$$\texttt{wf} :: \texttt{X} \rightarrow \texttt{Bool}$$

For a given value $x :: \texttt{X}$, $\texttt{wf}\ x$ returns true iff our data invariants hold for the value x.

Properties

For each operation, if all input values of type `X` satisfy `wf`, all output values will satisfy `wf`.
In other words, for each constructor operation $c :: \cdots \rightarrow \texttt{X}$, we must show $\texttt{wf}\ (c\ \ \cdots)$, and for each update operation $u :: \texttt{X} \rightarrow \texttt{X}$ we must show $\texttt{wf}\ x \implies \texttt{wf}(u\ x)$

Demo: Dictionary example, sorted order.

Stopping External Tampering

Even with our sorted dictionary example, there's nothing to stop a malicious or clueless programmer from going in and mucking up our data invariants.

Example

The malicious programmer could just add a word directly to the dictionary, unsorted, bypassing our carefully written `insert` function.

We want to prevent this sort of thing from happening.

Abstract Data Types

An *abstract* data type (ADT) is a data type where the implementation details of the type and its associated operations are hidden.

```
newtype Dict
type Word = String
type Definition = String
emptyDict  :: Dict
insertWord :: Word -> Definition -> Dict -> Dict
lookup     :: Word -> Dict -> Maybe Definition
```

If we don't have access to the implementation of `Dict`, then we can only access it via the provided operations, which we know preserve our data invariants. Thus, our data invariants cannot be violated if this module is correct.

Demo: In Haskell, we make ADTs with module headers.

Abstract? Data Types

In general, *abstraction* is the process of eliminating detail.

The inverse of abstraction is called *refinement*.

Abstract data types like the dictionary above are abstract in the sense that their implementation details are hidden, and we no longer have to reason about them on the level of implementation.

Validation

Suppose we had a sendEmail function

```
sendEmail :: String -- email address
          -> String -- message
          -> IO ()  -- action (more in 2 wks)
```

It is possible to mix the two `String` arguments, and even if we get the order right, it's possible that the given email address is not valid.

Question

Suppose that we wanted to make it impossible to call `sendEmail` without first checking that the email address was valid.
How would we accomplish this?

Validation ADTs

We could define a tiny ADT for validated email addresses, where the data invariant is that the contained email address is valid:

```
module EmailADT(Email, checkEmail, sendEmail)
    newtype Email = Email String

    checkEmail :: String -> Maybe Email
    checkEmail str | '@' `elem` str = Just (Email str)
                   | otherwise      = Nothing
```

Then, change the type of `sendEmail`:

```
    sendEmail :: Email -> String -> IO()
```

The only way (outside of the `EmailADT` module) to create a value of type `Email` is to use `checkEmail`.

`checkEmail` is an example of what we call a *smart constructor*: a constructor that enforces data invariants.

Reasoning about ADTs

Consider the following, more traditional example of an ADT interface, the unbounded queue:

```haskell
data Queue

emptyQueue :: Queue
enqueue :: Int -> Queue -> Queue
front   :: Queue -> Int    -- partial
dequeue :: Queue -> Queue  -- partial
size    :: Queue -> Int
```

We could try to come up with properties that relate these functions to each other without reference to their implementation, such as:

```
dequeue (enqueue x emptyQueue) == emptyQueue
```

However these do not capture functional correctness (usually).

Models for ADTs

We could imagine a simple implementation for queues, just in terms of lists:

```
emptyQueueL  = []
enqueueL a   = (++ [a])
frontL       = head
dequeueL     = tail
sizeL        = length
```

But this implementation is $\mathcal{O}(n)$ to enqueue! Unacceptable!

However!

This is a dead simple implementation, and trivial to see that it is correct. If we make a better queue implementation, it should always give the same results as this simple one. Therefore: This implementation serves as a functional correctness specification for our `Queue` type!

Refinement Relations

The typical approach to connect our model queue to our `Queue` type is to define a relation, called a *refinement relation*, that relates a `Queue` to a list and tells us if the two structures represent the same queue conceptually:

```
rel :: Queue -> [Int] -> Bool
```

Then, we show that the refinement relation holds initially:

```
prop_empty_r = rel emptyQueue emptyQueueL
```

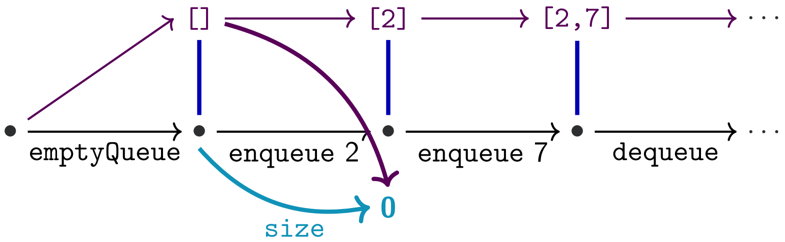
That any query functions for our two types produce equal results for related inputs, such as for size:

```
prop_size_r fq lq = rel fq lq ==> size fq == sizeL lq
```

And that each of the queue operations preserves our refinement relation, for example for enqueue:

```
prop_enq_ref fq lq x =
  rel fq lq ==> rel (enqueue x fq) (enqueueL x lq)
```

In Pictures




```
prop_enq_ref fq lq x =
rel fq lq ==> ref (enqueue x fq) (enqueueL x lq)
prop_empty_r = rel emptyQueue emptyQueueL
prop_size_r fq lq = rel fq lq ==> size fq == sizeL lq
```

Whenever we use a Queue, we can reason as if it were a list!

Abstraction Functions

These refinement relations are very difficult to use with QuickCheck because the `rel fq lq` preconditions are very hard to satisfy with randomly generated inputs.
For this example, it's a lot easier if we define an abstraction function that computes the corresponding abstract list from the concrete Queue.

$$\texttt{toAbstract} :: \texttt{Queue} \rightarrow [\texttt{Int}]$$

Conceptually, our refinement relation is then just:

$$\backslash\texttt{fq lq} \rightarrow \texttt{absfun fq == lq}$$

However, we can re-express our properties in a much more QC-friendly format (**Demo**)

Fast Queues

Let's use test-driven development! We'll implement a fast `Queue` with amortised $\mathcal{O}(1)$ operations.

```
data Queue = Q [Int] -- front of the queue
               Int   -- size of the front
               [Int] -- rear of the queue
               Int   -- size of the rear
```

We store the rear part of the queue in reverse order, to make enqueueing easier.

Thus, converting from our `Queue` to an abstract list requires us to reverse the rear:

```
toAbstract :: Queue -> [Int]
toAbstract (Q f sf r sr) = f ++ reverse r
```

Data Refinement

These kinds of properties establish what is known as a *data refinement* from the abstract, slow, list model to the fast, concrete `Queue` implementation.

Refinement and Specifications

In general, all functional correctness specifications can be expressed as:

1. all data invariants are maintained, and
2. the implementation is a refinement of an abstract correctness model.

There is a limit to the amount of abstraction we can do before they become useless for testing (but not necessarily for proving).

Warning

While abstraction can simplify proofs, abstraction does not reduce the fundamental complexity of verification, which is provably hard.

Data Invariants for Queue

In addition to the already-stated refinement properties, we also have some data invariants to maintain for a value `Q f sf r sr`:

1. `length f == sf`
2. `length r == sr`
3. **important**: `sf` $\geq$ `sr` — the front of the queue cannot be shorter than the rear.

We will ensure our `Arbitrary` instance only ever generates values that meet these invariants.

Thus, our `wellformed` predicate is used merely to enforce these data invariants on the outputs of our operations:

```
prop_wf_empty = wellformed (emptyQueue)
prop_wf_enq q = wellformed (enqueue x q)
prop_wf_deq q = size q > 0 ==> wellformed (dequeue q)
```

Implementing the Queue

We will generally implement by:

- Dequeue from the front.
- Enqueue to the rear.
- If necessary, re-establish the third data invariant by taking the rear, reversing it, and appending it to the front.

  This step is slow ($\mathcal{O}(n)$), but only happens every n operations or so, giving an average case amortised complexity of $\mathcal{O}(1)$ time.

Amortised Cost

```
enqueue x (Q f sf r sr) = inv3 (Q f sf (x:r) (sr + 1))
```

When we enqueue each of $[1..7]$ to the `emptyQueue` in turn:

$$
\begin{array}{llllll}
 & \texttt{Q} & [] & 0 & [] & 0 & \\
\rightarrow & \texttt{Q} & [1] & 1 & [] & 0 & (*) \\
\rightarrow & \texttt{Q} & [1] & 1 & [2] & 1 & \\
\rightarrow & \texttt{Q} & [1,2,3] & 3 & [] & 0 & (*) \\
\rightarrow & \texttt{Q} & [1,2,3] & 3 & [4] & 1 & \\
\rightarrow & \texttt{Q} & [1,2,3] & 3 & [5,4] & 2 & \\
\rightarrow & \texttt{Q} & [1,2,3] & 3 & [6,5,4] & 3 & \\
\rightarrow & \texttt{Q} & [1,2,3,4,5,6,7] & 7 & [] & 0 & (*)
\end{array}
$$

Observe that the slow invariant-reestablishing step $(*)$ happens after 1 step, then 2, then 4...

Extended out, this averages out to $\mathcal{O}(1)$.

Another Example

Consider this ADT interface for a bag of numbers:

```
data Bag
emptyBag   :: Bag
addToBag   :: Int -> Bag -> Bag
averageBag :: Bag -> Maybe Int
```

Our conceptual abstract model is just a list of numbers:

```
emptyBagA = []

addToBagA x xs = x:xs

averageBagA [] = Nothing
averageBagA xs = Just (sum xs `div` length xs)
```

But do we need to keep track of all that information in our implementation? No!

Concrete Implementation

Our concrete version will just maintain two integers, the `total` and the `count`:

```
data Bag = B { total :: Int , count :: Int }
emptyBag :: Bag
emptyBag = B 0 0

addToBag :: Int -> Bag -> Bag
addToBag x (B t c) = B (x + t) (c + 1)

averageBag :: Bag -> Maybe Int
averageBag (B _ 0) = Nothing
averageBag (B t c) = Just (t `div` c)
```

Refinement Functions

Normally, writing an abstraction function (as we did for `Queue`) is a good way to express our refinement relation in a QC-friendly way. In this case, however, it's hard to write such a function:

```
toAbstract :: Bag -> [Int]
toAbstract (B t c) = ?????
```

Instead, we will go in the other direction, giving us a *refinement function*:

```
toConc :: [Int] -> Bag
toConc xs = B (sum xs) (length xs)
```

Properties with Refinement Functions

Refinement functions produce properties much like abstraction functions, only with the abstract and concrete layers swapped:

```
prop_ref_empty =
   toConc emptyBagA == emptyBag

prop_ref_add x ab =
   toConc (addToBagA x ab) == addToBag x (toConc ab)

prop_ref_avg ab =
   averageBagA ab == averageBag (toConc ab)
```

Assignment 1 and Break

Assignment 1 has been **released**.
It is due right before the **Wednesday Lecture** of **Week 5**.

Advice from Alumni

The assignments do not involve much coding, but they do involve a lot of thinking.
Start early!

Homework

1. Get started on Assignment 1.
2. Next programming exercise is out, due before Friday 12pm.
3. Last week's quiz is due this Friday 6pm. Make sure you submit your answers.
4. This week's quiz is also up, due the following Friday.

COMP3141

Software System Design and Implementation

Effects and IO Monad Practice

Curtis Millar
CSE, UNSW

2 July 2021

QuickCheck and Search Trees

1. `mysteryProp`
2. `mysterious`
3. `astonishing`

Sequential composition

A function of the type `a -> (a, b)` is an *operation* that will produce a value of type `b` by *mutating* a value of type `a`.
If I have another function of type `a -> (a, c)`, I can *sequentially compose* them together. I can mutate the value of type `a` using the first function, then the second, and return the result of the second.

```
compose :: (a -> (a, b)) -> (a -> (a, c)) -> (a -> (a, c))
compose first second input =
  let
    (afterFirst, \_) = first input
  in
    second afterFirst
```

We compose the functions together by appying the first and discarding the resulting value of type `b`.

Using the output of an operation

If we determine our second operation based on the *result* of the first operation, we can represent it with a function of the type `b -> (a -> (a, c))`.
It is a function that consumes the result of a previous operation and produces a new operation. We can then create a function to sequence these operations together.

```
bind :: (a -> (a, b)) -> (b -> (a -> (a, c))) -> (a -> (a, c))
bind first second input =
  let
    (afterFirst, firstResult) = first input
  in
    second firstResult afterFirst
```

This is similar to `compose`, but rather than throw away the value of type `b`, we apply the function that produces the next opreation to that value.

State Monads

What we have in fact constructed is the `State` monad.

```haskell
newtype State s a = State (s -> (s, a))
```

The `State` monad encapsulates functions that mutate a state value and produce a result of that mutation.

The `State` monad has an instance of the `Monad` type class (we'll discuss further in week 7), so uses the >> operator rather than `compose` and the >>= operator rather than the `bind` function.

```haskell
(>>) :: Monad m => m a -> m b -> m b
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

We also get a `return` function that creates an operation that does nothing and returns a constant value.

```haskell
return :: Monad m => a -> m a
```

Do notation

As writing expressions using the >> and >>= operators can be tedious, Haskell provides do notation.

```haskell
-- Using >> operator
qux = foo >> bar

-- Using do
qux = do
  { foo
  ; bar
  }
```

```haskell
-- Using >>= operator
qux = foo >>= (\result -> bar)

-- Using do (without {;})
qux = do
  result <- foo
  bar
```

Functions on State Monads

The `State` monad provides three functions to interact with the internal state, `get`, `put`, and `modify`.

```haskell
-- Get the current state
get :: State s s
get = State $ (s -> (s, s))

-- Update the state
put :: s -> State s ()
put newState =
  State $ (_ -> (newState, ()))
```

```haskell
-- Mutate the state
modify :: (s -> s) -> State s ()
modify f = do
  s <- get
  put $ f s

-- Execute from starting state
runState :: State s a -> s -> (s, a)
runState (State op) = op
```

Mini Processor

Example

Mini processor done in editor

The IO Type

A procedure that performs some side effects, returning a result of type a is written as `IO a`.

World interpretation

`IO a` is an abstract type. But we can think of it as a function:

```
RealWorld -> (RealWorld, a)
```

(that's how it's implemented in GHC)

```haskell
(>>=) :: IO a -> (a -> IO b) -> IO b
pure  :: a -> IO a

getChar :: IO Char
readLine :: IO String
putStrLn :: String -> IO ()
readFile :: Strig -> IO String
```

QuickChecking Monads

QuickCheck lets us test `IO` (and `ST`) using this special property monad interface:

```
monadicIO :: PropertyM IO () -> Property
pre       :: Bool -> PropertyM IO ()
assert    :: Bool -> PropertyM IO ()
run       :: IO a -> PropertyM IO a
```

Testing a Tic-Tac-Toe A.I

Example

Testing A.Is for Tic-Tac-Toe

Done in editor

Homework

1. Next week is flexibility week
2. Last week's quiz is due on Today. Make sure you submit your answers.
3. The fourth programming exercise is due by the start if my next lecture (in 14 days).
4. This week's quiz is also up, it's due Friday week (in 14 days).

COMP3141

Software System Design and Implementation

Effects and State

Christine Rizkallah
UNSW Sydney
Term 2 2021

Effects

Effects

Effects are observable phenomena from the execution of a program.

Example (Memory effects)

```
int *p = ...
... // read and write
*p = *p + 1;
```

Example (IO)

```
// console IO
c = getchar();
printf("%d",32);
```

Example (Non-termination)

```
// infinite loop
while (1) {};
```

Example (Control flow)

```
// exception effect
throw new Exception();
```

Internal vs. External Effects

External Observability

An *external* effect is an effect that is observable outside the function.
Internal effects are not observable from outside.

Example (External effects)

Console, file and network I/O; termination and non-termination; non-local control flow; etc.

Are memory effects *external* or *internal*?

Answer: Depends on the scope of the memory being accessed. Global variable accesses are *external*.

Purity

A function with no external effects is called a *pure* function.

Pure functions

A *pure function* is the mathematical notion of a function. That is, a function of type `a -> b` is *fully* specified by a mapping from all elements of the domain type `a` to the codomain type `b`.

Consequences:

- Two invocations with the same arguments result in the same value.
- No observable trace is left beyond the result of the function.
- No implicit notion of time or order of execution.

Question: Are Haskell functions pure?

Haskell Functions

Haskell functions are technically not pure.

- They can loop infinitely.
- They can throw exceptions (partial functions).
- They can force evaluation of unevaluated expressions.

Caveat

Purity only applies to a particular level of abstraction. Even ignoring the above, assembly instructions produced by GHC aren't really pure.

Despite the impurity of Haskell functions, we can often reason as though they are pure. Hence we call Haskell a purely functional language.

The Danger of Implicit Side Effects

- They introduce (often subtle) requirements on the evaluation order.
- They are not visible from the type signature of the function.
- They introduce non-local dependencies which is bad for software design, increasing *coupling*.
- They interfere badly with strong typing, for example mutable arrays in Java, or reference types in ML.

We can't, in general, reason equationally about effectful programs!

Can we program with pure functions?

Yes! We've been doing it for the past 4 weeks.

Typically, a computation involving some state of type `s` and returning a result of type `a` can be expressed as a function:

```
s -> (s, a)
```

Rather than change the state, we return a new copy of the state.

Efficiency?

All that copying might seem expensive, but by using tree data structures, we can usually reduce the cost to an $\mathcal{O}(\log n)$ overhead.

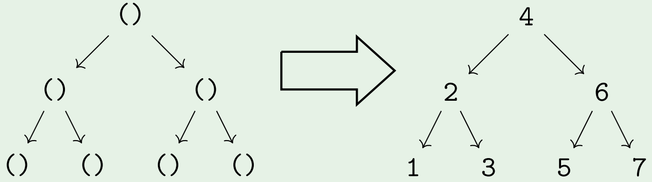
State Passing

Example (Labelling Nodes)

```
data Tree a = Branch a (Tree a) (Tree a) | Leaf
```

Given a tree, label each node with an ascending number in infix order:

```
label :: Tree () -> Tree Int
```



Let's use a data type to simplify this!

State

`newtype State s a` = A procedure that, manipulating some state of type `s`, returns `a`

State Operations

```
get :: State s s
put :: s -> State s ()
pure :: a -> State s a
evalState :: State s a -> s -> a
```

Sequential Composition

Do one state action after another with `do` blocks:

```
do put 42                 desugars     put 42 >> put True
   pure True                ==>
```

```
(>>) :: State s a -> State s b -> State s b
```

Example

Implement `modify`:

```
(s -> s) -> State s ()
```

And re-do the tree labelling.

Bind

The 2nd step can depend on the first with bind:

```
do x <- get               desugars     get >>= \x -> pure (x + 1)
   pure (x+1)               ==>
```

```
(>>=) :: State s a -> (a -> State s b) -> State s b
```

State Implementation

The `State` type is essentially implemented as the same state-passing we did before!

```haskell
newtype State s a = State (s -> (s,a))
```

Example

Let's implement each of the `State` operations for this newtype.

Caution

In the Haskell standard library `mtl`, the `State` type is actually implemented slightly differently, but the implementation essentially works the same way.

Effects

Sometimes we need side effects.

- We need to perform I/O, to communicate with the user or hardware.
- We might need effects for maximum efficiency.
  (but usually internal effects are sufficient)

Haskell's approach

Pure by default. Effectful when necessary.

The IO Type

A procedure that performs some side effects, returning a result of type a is written as `IO a`.

World interpretation

`IO a` is an abstract type. But we can think of it as a function:

```
RealWorld -> (RealWorld, a)
```

(that's how it's implemented in GHC)

```
(>>=) :: IO a -> (a -> IO b) -> IO b
pure  :: a -> IO a

getChar :: IO Char
readLine :: IO String
putStrLn :: String -> IO ()
```

Infectious IO

We can convert pure values to impure procedures with `pure`:

```
pure :: a -> IO a
```

But we can't convert impure procedures to pure values:

```
???? :: IO a -> a
```

The only function that gets an a from an `IO a` is `>>=`:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

But it returns an `IO` procedure as well.

Conclusion

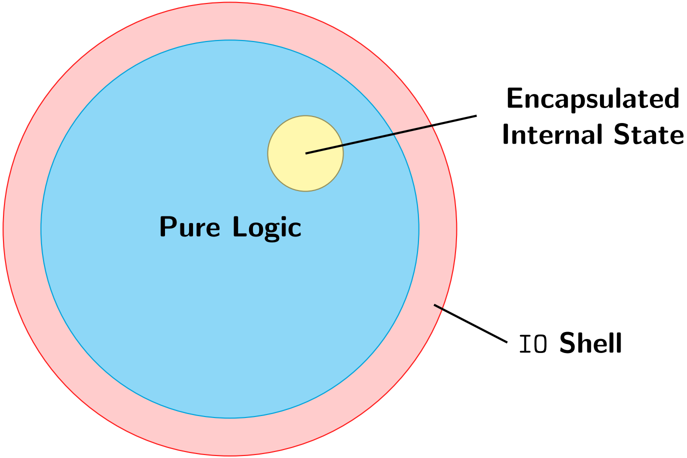
The moment you use an `IO` procedure in a function, `IO` shows up in the types, and you can't get rid of it!

If a function makes use of `IO` effects directly or indirectly, it will have `IO` in its type!

Haskell Design Strategy

We ultimately "run" `IO` procedures by calling them from `main`:

```
main :: IO ()
```



Examples

Example (Triangles)

Given an input number n, print a triangle of $*$ characters of base width n.

Example (Maze Game)

Design a game that reads in a $n \times n$ maze from a file. The player starts at position $(0,0)$ and must reach position $(n-1, n-1)$ to win. The game accepts keyboard input to move the player around the maze.

Benefits of an IO Type

- Absence of effects makes type system more informative:
  - A type signatures captures entire interface of the function.
  - All dependencies are explicit in the form of data dependencies.
  - All dependencies are typed.
- It is easier to reason about pure code and it is easier to test:
  - Testing is local, doesn't require complex set-up and tear-down.
  - Reasoning is local, doesn't require state invariants.
  - Type checking leads to strong guarantees.

Mutable Variables

We can have honest-to-goodness mutability in Haskell, if we really need it, using `IORef`.

```
data IORef a
newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

Example (Effectful Average)

Average a list of numbers using `IORef`s.

Mutable Variables, Locally

Something like averaging a list of numbers doesn't require external effects, even if we use mutation internally.

```
data STRef s a
newSTRef :: a -> ST (STRef s a)
readSTRef :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()
runST :: (forall s. ST s a) -> a
```

The extra `s` parameter is called a state thread, that ensures that mutable variables don't leak outside of the ST computation.

Note

The ST type is not assessable in this course, but it is useful sometimes in Haskell programming.

QuickChecking Effects

QuickCheck lets us test `IO` (and `ST`) using this special property monad interface:

```
monadicIO :: PropertyM IO () -> Property
pre       :: Bool -> PropertyM IO ()
assert    :: Bool -> PropertyM IO ()
run       :: IO a -> PropertyM IO a
```

Do notation and similar can be used for `PropertyM IO` procedures just as with `State s` and `IO` procedures.

Example (Testing `average`)

Let's test that our `IO` average function works like the non-effectful one.

Example (Testing `gfactor`)

Let's test that the GNU `factor` program works correctly!

Homework

1. New exercise out, due the week after next week.
2. Last week's quiz is due on Friday.
3. This week's quiz is due the Friday after the following Friday.

COMP3141

Software System Design and Implementation

Functors, Applicatives, and Monads Practice

Curtis Millar
CSE, UNSW
16 July 2021

Exercise 4

- Filter non-alphabetic characters from a file.
- Product of all numbers in the input file.
- Implement a guessing game AI.

State & IO

Week 5 covered `State` and `IO` and you've had a few weeks to work with them.
Do you have any questions?

Functors, Applicatives, Monads

- Consider *higher-kinded* types of kind `* -> *` that *contain* or *produce* their argument type.
- *Functor* lets us use a pure function to map between the higher-kinded type applied to different concrete types.
- *Applicative* lets us apply a n-ary function in the context of the higher-kinded type.
- *Monad* lets us *sequentially compose* functions that return values in the higher-kinded type.

Functors

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

The functor type class must obey two laws:

Functor Laws

1. `fmap id == id`
2. `fmap f . fmap g == fmap (f . g)`

Applicatives

```haskell
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

The functor type class must obey four additional laws:

Applicative Laws

1. `pure id <*> v = v` (Identity)
2. `pure f <*> pure x = pure (f x)` (Homomorphism)
3. `u <*> pure y = pure ($ y) <*> u` (Interchange)
4. `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)` (Composition)

Alternative Applicative

It is possible to express `Applicative` equivalently as:

```
class Functor f => App f where
  pure :: a -> f a
  tuple :: f a -> f b -> f (a,b)
```

Example (Alternative Applicative)

1. Using `tuple`, `fmap` and `pure`, let's implement `<*>`.
2. And, using `<*>`, `fmap` and `pure`, let's implement `tuple`.

done in Haskell.

Proof exercise: Prove that `tuple` obeys the applicative laws.

Monads

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
```

We can define a composition operator with `(>>=)`:

```
(<=<) :: (b -> m c) -> (a -> m b) -> (a -> m c)
(f <=< g) x = g x >>= f
```

The monad type class must obey three additional laws:

Monad Laws

1. `f <=< (g <=< x) == (f <=< g) <=< x` (associativity)
2. `pure <=< f == f` (left identity)
3. `f <=< pure == f` (right identity)

Alternative Monad

It is possible to express `Monad` equivalently as:

```
class Applicative m => Mon m where
  join :: m (m a) -> m a
```

Example (Alternative Monad)

1. Using `join` and `fmap`, let's implement >>=.
2. And, using >>= let's implement `join`.

done in Haskell.

Tree Example

```haskell
data Tree a
    = Leaf
    | Node a (Tree a) (Tree a)
     deriving (Show)
```

Example (Tree Example)

Show that `Tree` is an `Applicative` instance.
done in Haskell.

Note that `Tree` is not a `Monad` instance.

Homework

1. Week 5's quiz is due Today. Make sure you submit your answers.
2. The fifth programming exercise is due by the start of my next lecture (in 7 days).
3. This week's quiz is also up, it's due Friday week (in 7 days).

Consultations

- Consultations will be made on request. Ask on the forum or email `cs3141@cse.unsw.edu.au`.
- If there is a consultation it will be announced on the forum with a link a room number for Hopper.
- Will be in the Thursday lecture slot, 9am to 11am on Blackboard Collaborate.
- Make sure to join the queue on Hopper. Be ready to share your screen with REPL (`ghci` or `stack repl`) and editor set up.

COMP3141

Software System Design and Implementation

Functors, Applicatives, and Monads

Christine Rizkallah
UNSW Sydney
Term 2 2021

Motivation

We'll be looking at three very common abstractions:

- used in functional programming and,
- increasingly, in imperative programming as well.

Unlike many other languages, these abstractions are reified into bona fide type classes in Haskell, where they are often left as mere "design patterns" in other programming languages.

Kinds

Recall that terms in the type level language of Haskell are given *kinds*.
The most basic kind is written as `*`.

- Types such as `Int` and `Bool` have kind `*`.
- Seeing as `Maybe` is parameterised by one argument, `Maybe` has kind `* -> *`: given a type (e.g. `Int`), it will return a type (`Maybe Int`).

Question: What's the kind of `State`?

Functor

Recall the type class defined over type constructors called `Functor`.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Functor Laws

1. `fmap id == id`
2. `fmap f . fmap g == fmap (f . g)`

We've seen instances for lists, `Maybe`, tuples and functions.
Other instances include:

- `IO` (how?)
- `State s` (how?)
- `Gen`

Demonstrate in live-coding

QuickCheck Generators

Recall the `Arbitrary` class has a function:

```haskell
arbitrary :: Gen a
```

The type `Gen` is an abstract type for QuickCheck generators. Suppose we have a function:

```haskell
toString :: Int -> String
```

And we want a generator for `String` (i.e. `Gen String`) that is the result of applying `toString` to arbitrary `Int`s.
Then we use `fmap`!

Binary Functions

Suppose we want to look up a student's zID and program code using these functions:

```
lookupID :: Name -> Maybe ZID
lookupProgram :: Name -> Maybe Program
```

And we had a function:

```
makeRecord :: ZID -> Program -> StudentRecord
```

How can we combine these functions to get a function of type `Name -> Maybe StudentRecord`?

```
lookupRecord :: Name -> Maybe StudentRecord
lookupRecord n = let zid     = lookupID n
                     program = lookupProgram n
                  in ?
```

Binary Map?

We could imagine a binary version of the `maybeMap` function:

```
maybeMap2 :: (a -> b -> c)
          -> Maybe a -> Maybe b -> Maybe c
```

But then, we might need a trinary version.

```
maybeMap3 :: (a -> b -> c -> d)
          -> Maybe a -> Maybe b -> Maybe c -> Maybe d
```

Or even a 4-ary version, 5-ary, 6-ary. . .

this would quickly become impractical!

Using Functor

Using `fmap` gets us part of the way there:

```
lookupRecord' :: Name -> Maybe (Program -> StudentRecord)
lookupRecord' n = let zid     = lookupID n
                      program = lookupProgram n
                  in fmap makeRecord zid
                     -- what about program?
```

But, now we have a function inside a `Maybe`.

We need a function to take:

- A `Maybe`-wrapped fn `Maybe (Program -> StudentRecord)`
- A `Maybe`-wrapped argument `Maybe Program`

And apply the function to the argument, giving us a result of type `Maybe StudentRecord`?

Applicative

This is encapsulated by a subclass of `Functor` called `Applicative`:

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

`Maybe` is an instance, so we can use this for `lookupRecord`:

```
lookupRecord :: Name -> Maybe StudentRecord
lookupRecord n = let zid     = lookupID n
                     program = lookupProgram n
                  in fmap makeRecord zid <*> program
                -- or pure makeRecord <*> zid <*> program
```

Using Applicative

In general, we can take a regular function application:

```
f a b c d
```

And apply that function to `Maybe` (or other `Applicative`) arguments using this pattern (where `<*>` is left-associative):

```
pure f <*> ma <*> mb <*> mc <*> md
```

Relationship to Functor

All law-abiding instances of `Applicative` are also instances of `Functor`, by defining:

```
fmap f x = pure f <*> x
```

Sometimes this is written as an infix operator, `<$>`, which allows us to write:

```
pure f <*> ma <*> mb <*> mc <*> md
```

as:

```
f <$> ma <*> mb <*> mc <*> md
```

Proof exercise: From the applicative laws (next slide), prove that this implementation of `fmap` obeys the functor laws.

Applicative laws

```haskell
-- Identity
pure id <*> v = v

-- Homomorphism
pure f <*> pure x = pure (f x)

-- Interchange
u <*> pure y = pure ($ y) <*> u

-- Composition
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
```

These laws are a bit complex, and we certainly don't expect you to memorise them, but pay attention to them when defining instances!

Applicative Lists

There are two ways to implement `Applicative` for lists:

```
(<*>) :: [a -> b] -> [a] -> [b]
```

1. Apply each of the given functions to each of the given arguments, concatenating all the results.
2. Apply each function in the list of functions to the corresponding value in the list of arguments.

Question: How do we implement `pure`?

The second one is put behind a `newtype` (`ZipList`) in the Haskell standard library.

Other instances

- QuickCheck generators: `Gen`
  Recall from Wednesday Week 4:

```
data Concrete = C [Char] [Char]
  deriving (Show, Eq)

instance Arbitrary Concrete where
  arbitrary = C <$> arbitrary <*> arbitrary
```

- Functions: `((->) x)`
- Tuples: `((,) x)` We can't implement pure without an extra constraint!
- `IO` and `State s`:

On to Monads

- Functors are types for containers where we can `map` pure functions on their contents.
- Applicative Functors are types where we can combine n containers with a n-ary function.

The last and most commonly-used higher-kinded abstraction in Haskell programming is the `Monad`.

Monads

Monads are types `m` where we can *sequentially compose* functions of the form `a -> m b`

Monads

```haskell
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
```

Sometimes in old documentation the function `return` is included here, but it is just an alias for `pure`. It has nothing to do with `return` as in C/Java/Python etc.

Consider for:

- `Maybe`
- Lists
- `(x ->)` (the Reader monad)
- `(x,)` (the Writer monad, assuming a `Monoid` instance for `x`)
- `Gen`
- `IO`, `State s` etc.

Monad Laws

We can define a composition operator with (>>=):

```
(<=<) :: (b -> m c) -> (a -> m b) -> (a -> m c)
(f <=< g) x = g x >>= f
```

Monad Laws

```
f <=< (g <=< x) == (f <=< g) <=< x -- associativity
pure <=< f      == f               -- left identity
f <=< pure      == f               -- right identity
```

These are similar to the monoid laws, generalised for multiple types inside the monad. This sort of structure is called a *category* in mathematics.

Relationship to Applicative

All `Monad` instances give rise to an `Applicative` instance, because we can define `<*>` in terms of `>>=`.

```
mf <*> mx = mf >>= \f -> mx >>= \x -> pure (f x)
```

This implementation is already provided for `Monad`s as the ap function in `Control.Monad`.

Do notation

Working directly with the monad functions can be unpleasant.
As we've seen, Haskell has some notation to increase niceness:

```
do x <- y
   z
```

becomes

```
y >>= \x -> do z
```

```
do x
   y
```

becomes

```
x >>= \_ -> do y
```

We'll use this for most of our examples.

Examples

Example (Dice Rolls)

Roll two 6-sided dice, if the difference is < 2, reroll the second die. Final score is the difference of the two die. What score is most common?

Example (Partial Functions)

We have a list of student names in a database of type `[(ZID, Name)]`. Given a list of zID's, return a `Maybe [Name]`, where `Nothing` indicates that a zID could not be found.

Example (Arbitrary Instances)

Define a `Tree` type and a generator for search trees:

```
searchTrees :: Int -> Int -> Generator Tree
```

Homework

1. Next programming exercise is out now, due in Friday 12pm of Week 8.
2. This week's quiz is also up, due in Friday 6pm of Week 8.

COMP3141

Software System Design and Implementation

GADTs Practice

Curtis Millar
CSE, UNSW
23 July 2021

Exercise 5

- Parse a series of tokens.
- Stack push and pop.
- Evaluate a sequence of tokens.
- Calculate a string.

Recall: GADTs

Generalised Algebraic Datatypes (*GADTs*) is an extension to Haskell that, among other things, allows data types to be specified by writing the types of their constructors:

```haskell
{-# LANGUAGE GADTs, KindSignatures #-}
-- Unary natural numbers, e.g. 3 is S (S (S Z))
data Nat = Z | S Nat
-- is the same as
data Nat :: * where
  Z :: Nat
  S :: Nat -> Nat
```

The `printf` function

Consider the well known C function `printf`:

```
printf("Hello %s You are %d years old!", "Nina", 22)
```

In C, the type (and number) of parameters passed to this function are dependent on the first parameter (the format string).

The `printf` function

To do a similar thing in Haskell, we would like to use a richer type that allows us to define a function whose subsequent parameter is determined by the first.

```
{-# LANGUAGE StandaloneDeriving #-}
data Format :: * -> * where
  End :: Format ()                             -- Empty format
  Str :: Format t -> Format (String, t)  -- %s
  Dec :: Format t -> Format (Int, t)        -- %d
  L :: String -> Format t -> Format t  -- literal strings
-- just like deriving (Show) for normal data types.
deriving instance Show (Format ts)
```

Our format strings are indexed by a tuple type containing all of the types of the %directives used.

The `printf` function

```
"Hello %s You are %d years old!"
```

is written:

```
L "Hello" $ Str $ L " You are "
     $ Dec $ L " years old!" End
```

The printf function

```
printf :: Format ts -> ts -> IO ()
printf End () =
    pure () -- type is ()
printf (Str fmt) (s,ts) =
    do putStr s; printf fmt ts -- type is (String, ...)
printf (Dec fmt) (i,ts) =
    do putStr (show i); printf fmt ts -- type is (Int,..)
printf (L s fmt) ts      =
    do putStr s; printf fmt ts
```

Vectors

Define a natural number kind to use on the type level:

```haskell
data Nat = Z | S Nat
```

Our length-indexed list can be defined, called a `Vec`:

```haskell
{-# LANGUAGE DataKinds #-}
data Vec (a :: *) :: Nat -> * where
  Nil  :: Vec a Z
  Cons :: a -> Vec a n -> Vec a (S n)
```

The functions `hd` and `tl` can be total:

```haskell
hd :: Vec a (S n) -> a
hd (Cons x xs) = x
tl :: Vec a (S n) -> Vec a n
tl (Cons x xs) = xs
```

Vectors, continued

Our map for vectors is as follows:

```
mapVec :: (a -> b) -> Vec a n -> Vec b n
mapVec f Nil = Nil
mapVec f (Cons x xs) = Cons (f x) (mapVec f xs)
```

Properties

Using this type, it’s impossible to write a mapVec function that changes the length of the vector.

Properties are verified by the compiler!

Appending Vectors

Example (Problem)

```
appendV :: Vec a m -> Vec a n -> Vec a ???
```

We want to write `m + n` in the ??? above, but we do not have addition defined for kind `Nat`.

We can define a normal Haskell function easily enough:

```
plus :: Nat -> Nat -> Nat
plus Z y = y
plus (S x) y = S (plus x y)
```

This function is not applicable to type-level `Nat`s, though.

$\Rightarrow$ we need a type level function.

Type Families

Type level functions, also called *type families*, are defined in Haskell like so:

```
{-# LANGUAGE TypeFamilies #-}
type family Plus (x :: Nat) (y :: Nat) :: Nat where
  Plus Z     y = y
  Plus (S x) y = S (Plus x y)
```

We can use our type family to define appendV:

```
appendV :: Vec a m -> Vec a n -> Vec a (Plus m n)
appendV Nil         ys = ys
appendV (Cons x xs) ys = Cons x (appendV xs ys)
```

Concatenating Vectors

Example (Problem)

```
concatV :: Vec (Vec a m) n -> Vec a ???
```

We want to write `m * n` in the `???` above, but we do not have times defined for kind `Nat`.

```
{-# LANGUAGE TypeFamilies #-}
type family Times (a :: Nat) (b :: Nat) :: Nat where
  Times Z     m = Z
  Times (S n) m = Plus m (Times n m)
```

We can use our type family to define `concatV`:

```
{-# LANGUAGE UndecidableInstances #-}
concatV :: Vec (Vec a n) m -> Vec a (Times n m)
concatV Nil         = Nil
concatV (Cons v vs) = v `appendV` concatV vs
```

Filtering Vectors

Example (Problem)

```
filterV :: (a -> Bool) -> Vec a n -> Vec a ???
```

What is the size of the result of filter?

Filtering Vectors

Example (Problem)

```haskell
filterV :: (a -> Bool) -> Vec a n -> [a]
```

We do not know the size of the result.

We can use our type family to define `concatV`:

```haskell
filterV :: (a -> Bool) -> Vec a n -> [a]
filterV p Nil = []
filterV p (Cons x xs)
                  | p x   = x : filterV p xs
                  | otherwise = filterV p xs
```

Homework

1. Assignment 2 is **released**. The deadline has been extended; it is now due on **Wednesday 4th August, 6 PM** (in 12 days).
2. Week 7's quiz is due on today. Make sure you submit your answers.
3. The sixth programming exercise is due by the start of my next lecture (in 7 days).
4. This week's quiz is also up, it's due Friday next week at 6pm (in 7 days).

Consultations

- Consultations will be made on request. Ask on course forum or email `cs3141@cse.unsw.edu.au`.
- If there is a consultation it will be announced on the course forum with a link a room number for Hopper.
- Make sure to join the queue on Hopper. Be ready to share your screen with REPL (`ghci` or `stack repl`) and editor set up.

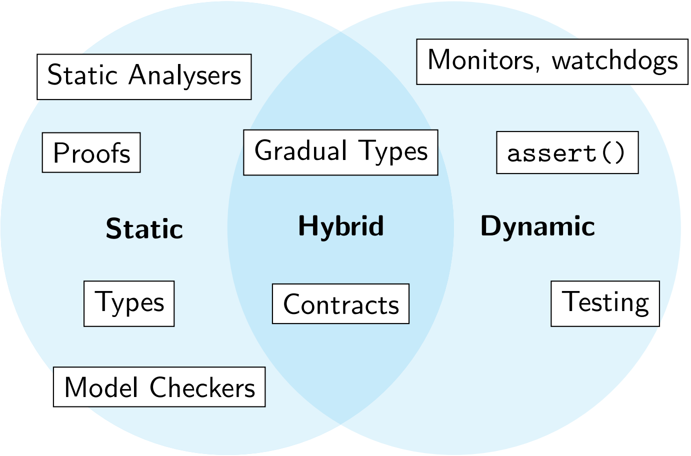
COMP3141

Software System Design and Implementation

Static Assurance with Types

Christine Rizkallah
UNSW Sydney
Term 2 2021

Methods of Assurance



Static means of assurance analyse a program without running it.

Static vs. Dynamic

- Static checks can be exhaustive.

Exhaustivity

An exhaustive check is a check that is able to analyse all possible executions of a program.

- **However**, some properties cannot be checked statically in general (halting problem), or are intractable to feasibly check statically (state space explosion).
- Dynamic checks cannot be exhaustive, but can be used to check some properties where static methods are unsuitable.

Compiler Integration

Most static and all dynamic methods of assurance are not integrated into the compilation process.

- You can compile and run your program even if it fails tests.
- You can change your program to diverge from your model checker model.
- Your proofs can diverge from your implementation.

Types

Because types are integrated into the compiler, they cannot diverge from the source code. This means that type signatures are a kind of machine-checked documentation for your code.

Types

Types are the most widely used kind of formal verification in programming today.

- They are checked automatically by the compiler.
- They can be extended to encompass properties and proof systems with very high expressivity (covered next week).
- They are an exhaustive analysis.

This week, we'll look at techniques to encode various correctness conditions inside Haskell's type system.

Phantom Types

Definition

A type parameter is *phantom* if it does not appear in the right hand side of the type definition.

```
newtype Size x = S Int
```

Let's examine each one of the following use cases:

- We can use this parameter to track what data invariants have been established about a value.
- We can use this parameter to track information about the representation (e.g. units of measure).
- We can use this parameter to enforce an ordering of operations performed on these values (*type state*).

Validation

```
data UG -- empty type
data PG
data StudentID x = SID Int
```

We can define a smart constructor that specialises the type parameter:

```
sid :: Int -> Either (StudentID UG)
                     (StudentID PG)
```

(Recalling the following definition of Either)

```
data Either a b = Left a | Right b
```

And then define functions:

```
enrolInCOMP3141 :: StudentID UG -> IO ()
lookupTranscript :: StudentID x -> IO String
```

Units of Measure

In 1999, software confusing units of measure (pounds and newtons) caused a mars orbiter to burn up on atmospheric entry.

```
data Kilometres
data Miles
data Value x = U Int
sydneyToMelbourne = (U 877 :: Value Kilometres)
losAngelesToSanFran = (U 383 :: Value Miles)
```

In addition to tagging values, we can also enforce constraints on units:

```
data Square a
area :: Value m -> Value m -> Value (Square m)
area (U x) (U y) = U (x * y)
```

Note the arguments to area must have the same units.

Type State

Example

A Socket can either be ready to recieve data, or busy. If the socket is busy, the user must first use the wait operation, which blocks until the socket is ready. If the socket is ready, the user can use the send operation to send string data, which will make the socket busy again.

```
data Busy
data Ready
newtype Socket s = Socket ...

wait :: Socket Busy -> IO (Socket Ready)

send :: Socket Ready -> String -> IO (Socket Busy)
```

What assumptions are we making here?

Linearity and Type State

The previous code assumed that we didn't re-use old `Socket`s:

```
send2 :: Socket Ready -> String -> String
      -> IO (Socket Busy)
send2 s x y = do s'   <- send s x
                 s''  <- wait s'
                 s''' <- send s'' y
                 pure s'''
```

But we can just re-use old values to send without waiting:

```
send2' s x y = do _  <- send s x
                  s' <- send s y
                  pure s'
```

Linear type systems can solve this, but not in Haskell (yet).

Datatype Promotion

```haskell
data UG
data PG
data StudentID x = SID Int
```

Defining empty data types for our tags is untyped. We can have `StudentID UG`, but also `StudentID String`.

Recall

Haskell types themselves have types, called kinds. Can we make the kind of our tag types more precise than $*$?

The `DataKinds` language extension lets us use data types as kinds:

```haskell
{-# LANGUAGE DataKinds, KindSignatures #-}
data Stream = UG | PG
data StudentID (x :: Stream) = SID Int
-- rest as before
```

Motivation: Evaluation

```haskell
data Expr = BConst Bool
          | IConst Int
          | Times Expr Expr
          | Less Expr Expr
          | And Expr Expr
          | If Expr Expr Expr
data Value = BVal Bool | IVal Int
```

Example

Define an expression evaluator:

```haskell
eval :: Expr -> Value
```

Motivation: Partiality

Unfortunately the `eval` function is partial, undefined for input expressions that are not well-typed, like:

```
And (ICons 3) (BConst True)
```

Recall

With any partial function, we can make it total by either expanding the co-domain (e.g. with a `Maybe` type), or constraining the domain.

Can we use phantom types to constrain the domain of `eval` to only accept well-typed expressions?

Attempt: Phantom Types

Let's try adding a phantom parameter to `Expr`, and defining typed constructors with precise types:

```
data Expr t = ...
bConst :: Bool -> Expr Bool
bConst = BConst
iConst :: Int -> Expr Int
iConst = IConst
times :: Expr Int -> Expr Int -> Expr Int
times = Times
less :: Expr Int -> Expr Int -> Expr Bool
less = Less
and :: Expr Bool -> Expr Bool -> Expr Bool
and = And
if'  :: Expr Bool -> Expr a -> Expr a -> Expr a
if' = If
```

Attempt: Phantom Types

This makes invalid expressions into type errors (yay!):

```
-- Couldn't match Int and Bool
and (iCons 3) (bConst True)
```

How about our `eval` function? What should its type be now?

```
eval :: Expr t -> t
```

Bad News

Inside `eval`, the Haskell type checker cannot be sure that we used our typed constructors, so in e.g. the `IConst` case:

```
eval :: Expr t -> t
eval (IConst i) = i -- type error
```

We are unable to tell that the type `t` is definitely `Int`.

Phantom types aren't strong enough!

GADTs

Generalised Algebraic Datatypes (*GADTs*) is an extension to Haskell that, among other things, allows data types to be specified by writing the types of their constructors:

```
{-# LANGUAGE GADTs, KindSignatures #-}
-- Unary natural numbers, e.g. 3 is S (S (S Z))
data Nat = Z | S Nat
-- is the same as
data Nat :: * where
  Z :: Nat
  S :: Nat -> Nat
```

When combined with the *type indexing* trick of phantom types, this becomes very powerful!

Expressions as a GADT

```haskell
data Expr :: * -> * where
    BConst :: Bool -> Expr Bool
    IConst :: Int -> Expr Int
    Times :: Expr Int -> Expr Int -> Expr Int
    Less :: Expr Int -> Expr Int -> Expr Bool
    And :: Expr Bool -> Expr Bool -> Expr Bool
    If  :: Expr Bool -> Expr a -> Expr a -> Expr a
```

Observation

There is now only *one* set of precisely-typed constructors.

Inside `eval` now, the Haskell type checker accepts our previously problematic case:

```haskell
eval :: Expr t -> t
eval (IConst i) = i -- OK now
```

GHC now knows that if we have `IConst`, the type `t` must be `Int`.

Lists

We could define our own list type using GADT syntax as follows:

```
data List (a :: *) :: * where
  Nil  :: List a
  Cons :: a -> List a -> List a
```

But, if we define head (`hd`) and tail (`tl`) functions, they're partial (boo!):

```
hd (Cons x xs) = x
tl (Cons x xs) = xs
```

We will constrain the domain of these functions by tracking the length of the list on the type level.

Vectors

As before, define a natural number kind to use on the type level:

```
data Nat = Z | S Nat
```

Now our length-indexed list can be defined, called a `Vec`:

```
data Vec (a :: *) :: Nat -> * where
  Nil  :: Vec a Z
  Cons :: a -> Vec a n -> Vec a (S n)
```

Now `hd` and `tl` can be total:

```
hd :: Vec a (S n) -> a
hd (Cons x xs) = x
tl :: Vec a (S n) -> Vec a n
tl (Cons x xs) = xs
```

Vectors, continued

Our `map` for vectors is as follows:

```
mapVec :: (a -> b) -> Vec a n -> Vec b n
mapVec f Nil = Nil
mapVec f (Cons x xs) = Cons (f x) (mapVec f xs)
```

Properties

Using this type, it's impossible to write a `mapVec` function that changes the length of the vector.

Properties are verified by the compiler!

Tradeoffs

The benefits of this extra static checking are obvious, however:

- It can be difficult to convince the Haskell type checker that your code is correct, even when it is.
- Type-level encodings can make types more verbose and programs harder to understand.
- Sometimes excessively detailed types can make type-checking very slow, hindering productivity.

Pragmatism

We should use type-based encodings only when the assurance advantages outweigh the clarity disadvantages.
The typical use case for these richly-typed structures is to eliminate partial functions from our code base.
If we never use partial list functions, length-indexed vectors are not particularly useful.

Appending Vectors

Example (Problem)

```
appendV :: Vec a m -> Vec a n -> Vec a ???
```

We want to write `m + n` in the `???` above, but we do not have addition defined for kind `Nat`.

We can define a normal Haskell function easily enough:

```
plus :: Nat -> Nat -> Nat
plus Z y = y
plus (S x) y = S (plus x y)
```

This function is not applicable to type-level `Nat`s, though.

$\Rightarrow$ we need a type level function.

Type Families

Type level functions, also called *type families*, are defined in Haskell like so:

```
{-# LANGUAGE TypeFamilies #-}
type family Plus (x :: Nat) (y :: Nat) :: Nat where
  Plus Z     y = y
  Plus (S x) y = S (Plus x y)
```

We can use our type family to define `appendV`:

```
appendV :: Vec a m -> Vec a n -> Vec a (Plus m n)
appendV Nil         ys = ys
appendV (Cons x xs) ys = Cons x (appendV xs ys)
```

Recursion

If we had implemented `Plus` by recursing on the second argument instead of the first:

```
{-# LANGUAGE TypeFamilies #-}
type family Plus' (x :: Nat) (y :: Nat) :: Nat where
  Plus' x Z     = x
  Plus' x (S y) = S (Plus' x y)
```

Then our appendV code would not type check.

```
appendV :: Vec a m -> Vec a n -> Vec a (Plus' m n)
appendV Nil         ys = ys
appendV (Cons x xs) ys = Cons x (appendV xs ys)
```

Why?

Answer

Consider the `Nil` case. We know `m = Z`, and must show that our desired return type `Plus′ Z n` equals our given return type `n`, but that fact is not immediately apparent from the equations.

Type-driven development

- This lecture is only a taste of the full power of type-based specifications.
- Languages supporting dependent types (Idris, Agda) completely merge the type and value level languages, and support machine-checked proofs about programs.
- Haskell is also gaining more of these typing features all the time.

Next week: Fancy theory about types!

- Deep connections between types, logic and proof.
- Algebraic type structure for generic algorithms and refactoring.
- Using polymorphic types to infer properties for free.

Participation Marks

1. As participation marks were not initially mentioned in the course outline, we decided to drop them.
2. Instead, we will provide one bonus mark to all students for the overall course mark, if we receive at least 50% course myExperience feedback. Another bonus point will be awarded to all students if we receive significant detailed feedback on the course forum (say 10% student response rate).

Homework and Marking

1. Some Assignment 1 submissions are not marked yet and are being review by UNSW's student conduct unit for alleged plagiarism. See Curtis' forum announcement for details on how this is handled.
2. Do not cheat. If you are struggling, ask for help.
3. Assignment 2 is **released**. Due on Friday **30th July, 12 PM**.
4. The last programming exercise has been released, also due next Friday at 12pm.
5. This week's quiz is also up, due in Friday of Week 9 at 6pm.

COMP3141

Software System Design and Implementation

More on the Curry Howard Isomorphism

Curtis Millar
CSE, UNSW
30 July 2021

Exercise 6

- Evaluate terms.
- Satisfiability.
- Enumerating solutions.

What is Intuitionistic Logic?

- Classical logic is the logic that most people know about.
- Intuitionistic logic does not contain the axiom of excluded middle $p \vee \neg p$ or equivalently $\neg\neg p \to p$.
- In classical logic more can be proven but less can be expressed.
- Intuitionistic proof of an existence statement gives a witness for the statement.

Example of Existence in the Classical Sense

- Let $\mathbb{Q}$ be the set of rational numbers and $\mathbb{I}$ be the set of irrational numbers.
- Consider the statement $\exists x, y.(x \in \mathbb{I}) \wedge (y \in \mathbb{I}) \wedge (x^y \in \mathbb{Q})$.
- Proof:
  - Consider the number $\sqrt{2}^{\sqrt{2}}$.

1. If $\sqrt{2}^{\sqrt{2}} \in \mathbb{Q}$
2. Otherwise if $\sqrt{2}^{\sqrt{2}} \in \mathbb{I}$

Example of Existence in the Classical Sense

- Let $\mathbb{Q}$ be the set of rational numbers and $\mathbb{I}$ be the set of irrational numbers.
- Consider the statement $\exists x, y.(x \in \mathbb{I}) \wedge (y \in \mathbb{I}) \wedge (x^y \in \mathbb{Q})$.
- Proof:
  - Consider the number $\sqrt{2}^{\sqrt{2}}$.

  1. If $\sqrt{2}^{\sqrt{2}} \in \mathbb{Q}$
     - Pick $x = \sqrt{2}$ and $y = \sqrt{2}$
     - Then $x^y = \sqrt{2}^{\sqrt{2}}$ so $x^y \in \mathbb{Q}$
  2. Otherwise if $\sqrt{2}^{\sqrt{2}} \in \mathbb{I}$
     -
     -

Example of Existence in the Classical Sense

- Let $\mathbb{Q}$ be the set of rational numbers and $\mathbb{I}$ be the set of irrational numbers.
- Consider the statement $\exists x, y.(x \in \mathbb{I}) \wedge (y \in \mathbb{I}) \wedge (x^y \in \mathbb{Q})$.
- Proof:
  - Consider the number $\sqrt{2}^{\sqrt{2}}$.

  1. If $\sqrt{2}^{\sqrt{2}} \in \mathbb{Q}$
     - Pick $x = \sqrt{2}$ and $y = \sqrt{2}$
     - Then $x^y = \sqrt{2}^{\sqrt{2}}$ so $x^y \in \mathbb{Q}$
  2. Otherwise if $\sqrt{2}^{\sqrt{2}} \in \mathbb{I}$
     - Pick $x = \sqrt{2}^{\sqrt{2}}$ and $y = \sqrt{2}$
     - Then $x^y = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^2 = 2$ so $x^y \in \mathbb{Q}$

Recall: The Curry-Howard Isomorphism

This correspondence goes by many names, but is usually attributed to Haskell Curry and William Howard.

It is a *very deep* result:

Logic	Programming
Propositions	Types
Proofs	Programs
Proof Simplification	Evaluation

It turns out, no matter what logic you want to define, there is always a corresponding λ-calculus, and vice versa.

Constructive Logic	Typed λ-Calculus
Classical Logic	Continuations
Modal Logic	Monads
Linear Logic	Linear Types, Session Types
Separation Logic	Region Types

Translating

We can translate logical connectives to types and back:

Conjunction ($\wedge$)	Tuples
Disjunction ($\vee$)	`Either`
Implication	Functions
True	`()`
False	`Void`

We can also translate our *equational reasoning* on programs into *proof simplification* on proofs!

Constructors and Eliminators for Sums

```haskell
data TrafficLight = Red | Amber | Green
```

Example (Traffic Lights)

$$\texttt{TrafficLight} \simeq \texttt{Either () (Either () ())}$$

$$\begin{aligned}
\texttt{Red} &\simeq \mathsf{Left}\ () \\
\texttt{Amber} &\simeq \mathsf{Right}\ (\mathsf{Left}\ ()) \\
\texttt{Green} &\simeq \mathsf{Right}\ (\mathsf{Right}\ (\mathsf{Left}\ ()))
\end{aligned}$$

Type Correctness

$$\frac{}{\Gamma \vdash \texttt{()} :: \texttt{()}}\,\texttt{()} \qquad \frac{\Gamma \vdash e :: A}{\Gamma \vdash \texttt{Left}\ e :: \texttt{Either}\ A\ B}\,\mathrm{S}_L \qquad \frac{\Gamma \vdash e :: B}{\Gamma \vdash \texttt{Right}\ e :: \texttt{Either}\ A\ B}\,\mathrm{S}_R$$

$$\frac{\dfrac{\dfrac{???}{\texttt{Left ()} :: \texttt{()}}}{\texttt{Right (Left ())} :: \texttt{Either () ()}}\,\mathrm{S}_R}{\texttt{Right (Right (Left ()))} :: \texttt{Either () (Either () ())}}\,\mathrm{S}_R$$

Type Correctness

$$\dfrac{}{\Gamma \vdash () :: ()}\,()
\qquad
\dfrac{\Gamma \vdash e :: A}{\Gamma \vdash \texttt{Left}\ e :: \texttt{Either}\ A\ B}\,\mathrm{S}_L
\qquad
\dfrac{\Gamma \vdash e :: B}{\Gamma \vdash \texttt{Right}\ e :: \texttt{Either}\ A\ B}\,\mathrm{S}_R$$

$$\dfrac{\dfrac{\dfrac{}{\texttt{()} :: \texttt{()}}\,()}{\texttt{Right ()} :: \texttt{Either () ()}}\,\mathrm{S}_R}{\texttt{Right (Right ())} :: \texttt{Either () (Either () ())}}\,\mathrm{S}_R$$

Examples

```haskell
prop_or_false :: a -> (Either a Void)
prop_or_false a = Left a

prop_or_true :: a -> (Either a ())
prop_or_true a = Right ()

prop_and_true :: a -> (a, ())
prop_and_true a = (a, ())

prop_double_neg_intro :: a -> (a -> Void) -> Void
prop_double_neg_intro a f = f a

prop_triple_neg_elim ::
  (((a-> Void) -> Void) -> Void) -> a -> Void
prop_triple_neg_elim f a = f (\g ->  g a)
```

Wrap-up

1. Assignment 2 is due Wednesday evening (4th August 6pm).
2. There is a quiz for this week, but no exercise.
3. Next week's lectures consist of an **extension or guest** lecture and a **revision lecture** on Friday.
4. Revision topics can be posted in the forum
5. If you enjoyed the course and want to do more in this direction, ask us for thesis topics, taste of research projects, and consider attending COMP3161 and COMP4161.
6. Fill in the myExperience reports, it is important for us to receive your feedback. Everyone gets bonus mark if we get over 50% response rate.

Consultations

- Consultations will be made on request. Ask on forum or email `cs3141@cse.unsw.edu.au`.
- If there is a consultation it will be announced on Piazza with a link a room number for Hopper.
- Will be on the Tuesday, 4pm to 5:30pm on Blackboard Collaborate.
- Make sure to join the queue on Hopper. Be ready to share your screen with REPL (`ghci` or `stack repl`) and editor set up.

COMP3141

Software System Design and Implementation

Theory of Types

Christine Rizkallah
UNSW Sydney

Term 2 2021

Natural Deduction

Logic

We can specify a logical system as a *deductive system* by providing a set of rules and axioms that describe how to prove properties about formulas involving various connectives.

Each connective typically has *introduction* and *elimination* rules.
For example, to prove an implication $A \to B$ holds, we must show that B holds assuming A. This introduction rule is written as:

derivability
(if the top, then the bottom)

$$\frac{A, \Gamma \vdash B}{\Gamma \vdash A \to B}\to\text{-I}$$

entailment
(assuming the left, we can prove the right)

More rules

Implication also has an elimination rule, that is also called *modus ponens*:

$$\frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\to\text{-E}$$

Conjunction (and) has an introduction rule that follows our intuition:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \land B}\land\text{-I}$$

It has two elimination rules:

$$\frac{\Gamma \vdash A \land B}{\Gamma \vdash A}\land\text{-E}_1 \qquad \frac{\Gamma \vdash A \land B}{\Gamma \vdash B}\land\text{-E}_2$$

More rules

Disjunction (or) has two introduction rules:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \lor B}\lor\text{-I}_1 \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \lor B}\lor\text{-I}_2$$

Disjunction elimination is a little unusual:

$$\frac{\Gamma \vdash A \lor B \qquad A, \Gamma \vdash P \qquad B, \Gamma \vdash P}{\Gamma \vdash P}\lor\text{-E}$$

The true literal, written $\top$, has only an introduction:

$$\frac{}{\Gamma \vdash \top}$$

And false, written $\bot$, has just elimination (*ex falso quodlibet*):

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash P}$$

Example Proofs

Example

Prove:

- $A \wedge B \to B \wedge A$
- $A \vee \bot \to A$

What would negation be equivalent to?
Typically we just define

$$\neg A \equiv (A \to \bot)$$

.

Example

Prove:

- $A \to (\neg\neg A)$
- $(\neg\neg A) \to A$ We get stuck here!

Constructive Logic

The logic we have expressed so far does not admit the law of the excluded middle:

$$P \vee \neg P$$

Or the equivalent double negation elimination:

$$(\neg\neg P) \rightarrow P$$

This is because it is a *constructive* logic that does not allow us to do proof by contradiction.

Boiling Haskell Down

The theoretical properties we will describe also apply to Haskell, but we need a smaller language for demonstration purposes.

- No user-defined types, just a small set of built-in types.
- No polymorphism (type variables)
- Just lambdas ($\lambda x.e$) to define functions or bind variables.

This language is a very minimal functional language, called the simply typed lambda calculus, originally due to Alonzo Church.

Our small set of built-in types are intended to be enough to express most of the data types we would otherwise define.
We are going to use logical inference rules to specify how expressions are given types (*typing rules*).

Function Types

We create values of a function type $A \rightarrow B$ using lambda expressions:

$$\frac{x :: A, \Gamma \vdash e :: B}{\Gamma \vdash \lambda x.\ e :: A \rightarrow B}$$

The typing rule for function application is as follows:

$$\frac{\Gamma \vdash e_1 :: A \rightarrow B \qquad \Gamma \vdash e_2 :: A}{\Gamma \vdash e_1\ e_2 :: B}$$

What other types would be needed?

Composite Data Types

In addition to functions, most programming languages feature ways to *compose* types together to produce new types, such as:

Classes

Tuples

Structs

Unions

Records

Combining values conjunctively

We want to store two things in one value.

(might want to use non-compact slides for this one)

types

Has

```
type Point
midpoint (
  = ((x1+x2
```

y2)

C Structs

```
typed
  flo
  flo
} poi
poin
  poi
  mid
  mid
  ret
}
```

Java

```
cl
}
Po
}
```

“Better” Java

```
class Point {
  private float x;
  private float y;
  public Point (float x, float y) {
      this.x = x; this.y = y;
  }
  public float getX() {return this.x;}
  public float getY() {return this.y;}
  public float setX(float x) {this.x=x;}
  public float setY(float y) {this.y=y;}
}
Point midPoint (Point p1, Point p2) {
  return new Point((p1.getX() + p2.getX()) / 2.0,
                   (p2.getY() + p2.getY()) / 2.0);
```

Product Types

For simply typed lambda calculus, we will accomplish this with tuples, also called *product types*.

$$(A, B)$$

We won't have type declarations, named fields or anything like that. More than two values can be combined by nesting products, for example a three dimensional vector:

$$(\texttt{Int}, (\texttt{Int}, \texttt{Int}))$$

Constructors and Eliminators

We can construct a product type the same as Haskell tuples:

$$\frac{\Gamma \vdash e_1 :: A \qquad \Gamma \vdash e_2 :: B}{\Gamma \vdash (e_1, e_2) :: (A, B)}$$

The only way to extract each component of the product is to use the `fst` and `snd` eliminators:

$$\frac{\Gamma \vdash e :: (A, B)}{\Gamma \vdash \mathsf{fst}\ e :: A} \qquad \frac{\Gamma \vdash e :: (A, B)}{\Gamma \vdash \mathsf{snd}\ e :: B}$$

Unit Types

Currently, we have no way to express a type with just one value. This may seem useless at first, but it becomes useful in combination with other types.

We'll introduce the unit type from Haskell, written `()`, which has exactly one inhabitant, also written `()`:

$$\frac{}{\Gamma \vdash () :: ()}$$

Disjunctive Composition

We can't, with the types we have, express a type with exactly three values.

Example (Trivalued type)

```
data TrafficLight = Red | Amber | Green
```

In general we want to express data that can be one of multiple alternatives, that contain different bits of data.

Example (More elaborate alternatives)

```
type Length = Int
type Angle = Int
data Shape = Rect Length Length
           | Circle Length | Point
           | Triangle Angle Length Length
```

This is awkward in many languages. In Java we'd have to use inheritance. In C we'd have to use unions.

Sum Types

We'll build in the Haskell `Either` type to express the possibility that data may be one of two forms.

$$\texttt{Either}\ A\ B$$

These types are also called *sum types*.

Our `TrafficLight` type can be expressed (grotesquely) as a sum of units:

$$\texttt{TrafficLight} \simeq \texttt{Either}\ ()\ (\texttt{Either}\ ()\ ())$$

Constructors and Eliminators for Sums

To make a value of type `Either` A B, we invoke one of the two constructors:

$$\frac{\Gamma \vdash e :: A}{\Gamma \vdash \mathsf{Left}\ e :: \mathtt{Either}\ A\ B} \qquad \frac{\Gamma \vdash e :: B}{\Gamma \vdash \mathsf{Right}\ e :: \mathtt{Either}\ A\ B}$$

We can branch based on which alternative is used using pattern matching:

$$\frac{\Gamma \vdash e :: \mathtt{Either}\ A\ B \qquad x :: A, \Gamma \vdash e_1 :: P \qquad y :: B, \Gamma \vdash e_2 :: P}{\Gamma \vdash (\mathbf{case}\ e\ \mathbf{of}\ \mathsf{Left}\ x \to e_1; \mathsf{Right}\ y \to e_2) :: P}$$

Examples

Example (Traffic Lights)

Our traffic light type has three values as required:

$$\texttt{TrafficLight} \simeq \texttt{Either () (Either () ())}$$

$$\begin{array}{lcl} \texttt{Red} & \simeq & \mathsf{Left}\ () \\ \texttt{Amber} & \simeq & \mathsf{Right}\ (\mathsf{Left}\ ()) \\ \texttt{Green} & \simeq & \mathsf{Right}\ (\mathsf{Right}\ ()) \end{array}$$

The Empty Type

We add another type, called `Void`, that has no inhabitants. Because it is empty, there is no way to construct it.
We do have a way to eliminate it, however:

$$\frac{\Gamma \vdash e :: \texttt{Void}}{\Gamma \vdash \mathsf{absurd}\ e :: P}$$

If I have a variable of the empty type in scope, we must be looking at an expression that will never be evaluated. Therefore, we can assign any type we like to this expression, because it will never be executed.

Gathering Rules

$$\frac{\Gamma \vdash e :: \mathtt{Void}}{\Gamma \vdash \mathsf{absurd}\ e :: P} \qquad \frac{}{\Gamma \vdash () :: ()}$$

$$\frac{\Gamma \vdash e :: A}{\Gamma \vdash \mathsf{Left}\ e :: \mathtt{Either}\ A\ B} \qquad \frac{\Gamma \vdash e :: B}{\Gamma \vdash \mathsf{Right}\ e :: \mathtt{Either}\ A\ B}$$

$$\frac{\Gamma \vdash e :: \mathtt{Either}\ A\ B \qquad x :: A, \Gamma \vdash e_1 :: P \qquad y :: B, \Gamma \vdash e_2 :: P}{\Gamma \vdash (\textbf{case}\ e\ \textbf{of}\ \mathsf{Left}\ x \rightarrow e_1; \mathsf{Right}\ y \rightarrow e_2) :: P}$$

$$\frac{\Gamma \vdash e_1 :: A \qquad \Gamma \vdash e_2 :: B}{\Gamma \vdash (e_1, e_2) :: (A, B)} \qquad \frac{\Gamma \vdash e :: (A, B)}{\Gamma \vdash \mathsf{fst}\ e :: A} \qquad \frac{\Gamma \vdash e :: (A, B)}{\Gamma \vdash \mathsf{snd}\ e :: B}$$

$$\frac{\Gamma \vdash e_1 :: A \rightarrow B \qquad \Gamma \vdash e_2 :: A}{\Gamma \vdash e_1\ e_2 :: B} \qquad \frac{x :: A, \Gamma \vdash e :: B}{\Gamma \vdash \lambda x.\ e :: A \rightarrow B}$$

Removing Terms...

$$\frac{\Gamma \vdash \texttt{Void}}{\Gamma \vdash P} \qquad \frac{}{\Gamma \vdash ()}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash \texttt{Either}\ A\ B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash \texttt{Either}\ A\ B}$$

$$\frac{\Gamma \vdash \texttt{Either}\ A\ B \qquad A, \Gamma \vdash P \qquad B, \Gamma \vdash P}{\Gamma \vdash P}$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash (A, B)} \qquad \frac{\Gamma \vdash (A, B)}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash (A, B)}{\Gamma \vdash B}$$

$$\frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B} \qquad \frac{A, \Gamma \vdash B}{\Gamma \vdash A \to B}$$

This looks exactly like constructive logic!
Constructing a program of a certain type, also creates a proof of a certain proposition.

The Curry-Howard Correspondence

This correspondence goes by many names, but is usually attributed to Haskell Curry and William Howard.

It is a *very deep* result:

Programming	**Logic**
Types	Propositions
Programs	Proofs
Evaluation	Proof Simplification

It turns out, no matter what logic you want to define, there is always a corresponding λ-calculus, and vice versa.

Typed λ-Calculus	Constructive Logic
Continuations	Classical Logic
Monads	Modal Logic
Linear Types, Session Types	Linear Logic
Region Types	Separation Logic

Examples

Example (Commutativity of Conjunction)

$$\begin{array}{l} \mathit{andComm} :: (A, B) \to (B, A) \\ \mathit{andComm}\ p = (\mathsf{snd}\ p, \mathsf{fst}\ p) \end{array}$$

This proves $A \land B \to B \land A$.

Example (Transitivity of Implication)

$$\begin{array}{l} \mathit{transitive} :: (A \to B) \to (B \to C) \to (A \to C) \\ \mathit{transitive}\ f\ g\ x = g\ (f\ x) \end{array}$$

Transitivity of implication is just function composition.

Translating

We can translate logical connectives to types and back:

Tuples	Conjunction ($\wedge$)
`Either`	Disjunction ($\vee$)
Functions	Implication
`()`	True
`Void`	False

We can also translate our *equational reasoning* on programs into *proof simplification* on proofs!

Proof Simplification

Assuming $A \wedge B$, we want to prove $B \wedge A$.
We have this unpleasant proof:

$$\dfrac{\dfrac{A \wedge B}{B} \qquad \dfrac{\dfrac{\dfrac{A \wedge B}{A} \qquad \dfrac{A \wedge B}{A}}{A \wedge A}}{A}}{B \wedge A}$$

Proof Simplification

Translating to types, we get:
Assuming $x :: (A, B)$, we want to construct (B, A).

$$\dfrac{\dfrac{x :: (A,B)}{\mathsf{snd}\ x :: B} \qquad \dfrac{\dfrac{\dfrac{x :: (A,B)}{\mathsf{fst}\ x :: A} \quad \dfrac{x :: (A,B)}{\mathsf{fst}\ x :: A}}{(\mathsf{fst}\ x, \mathsf{fst}\ x) :: (A,A)}}{\mathsf{snd}\ (\mathsf{fst}\ x, \mathsf{fst}\ x) :: A}}{(\mathsf{snd}\ x, \mathsf{snd}\ (\mathsf{fst}\ x, \mathsf{fst}\ x)) :: (B,A)}$$

We know that

$$(\mathsf{snd}\ x, \mathsf{snd}\ (\mathsf{fst}\ x, \mathsf{fst}\ x)) \quad = \quad (\mathsf{snd}\ x, \mathsf{fst}\ x)$$

Let's apply this simplification to our proof!

Proof Simplification

Assuming $x :: (A, B)$, we want to construct (B, A).

$$\dfrac{\dfrac{x :: (A, B)}{\mathsf{snd}\ x :: B} \qquad \dfrac{x :: (A, B)}{\mathsf{fst}\ x :: A}}{(\mathsf{snd}\ x, \mathsf{fst}\ x) :: (B, A)}$$

Back to logic:

$$\dfrac{\dfrac{A \wedge B}{B} \qquad \dfrac{A \wedge B}{A}}{B \wedge A}$$

Applications

As mentioned before, in dependently typed languages such as Agda and Idris, the distinction between value-level and type-level languages is removed, allowing us to refer to our program in types (i.e. propositions) and then construct programs of those types (i.e. proofs).
Generally, dependent types allow us to use rich types not just for programming, but also for verification via the Curry-Howard correspondence.

Caveats

All functions we define have to be total and terminating.
Otherwise we get an *inconsistent* logic that lets us prove false things:

$$\begin{aligned} &\mathit{proof}_1 :: \mathrm{P} = \mathrm{NP} \\ &\mathit{proof}_1 = \mathit{proof}_1 \end{aligned}$$

$$\begin{aligned} &\mathit{proof}_2 :: \mathrm{P} \neq \mathrm{NP} \\ &\mathit{proof}_2 = \mathit{proof}_2 \end{aligned}$$

Most common calculi correspond to constructive logic, not classical ones, so principles like the law of excluded middle or double negation elimination do not hold:

$$\neg\neg P \to P$$

Semiring Structure

These types we have defined form an algebraic structure called a *commutative semiring*.

Laws for `Either` and `Void`:

- Associativity: `Either` (`Either` A B) C $\simeq$ `Either` A (`Either` B C)
- Identity: `Either Void` $A \simeq A$
- Commutativity: `Either` A $B \simeq$ `Either` B A

Laws for tuples and `1`

- Associativity: $((A,B),C) \simeq (A,(B,C))$
- Identity: $((),A) \simeq A$
- Commutativity: $(A,B) \simeq (B,A)$

Combining the two:

- Distributivity: (A, `Either` B C) $\simeq$ `Either` (A,B) (A,C)
- Absorption: (`Void`, A) $\simeq$ `Void`

What does $\simeq$ mean here? It's more than logical equivalence.

Isomorphism

Two types A and B are *isomorphic*, written $A \simeq B$, if there exists a *bijection* between them. This means that for each value in A we can find a unique value in B and vice versa.

Example (Refactoring)

We can use this reasoning to simplify type definitions. For example:

```
data Switch = On Name Int
            | Off Name
```

Can be simplified to the isomorphic `(Name, Maybe Int)`.

Generic Programming

Representing data types generically as sums and products is the foundation for generic programming libraries such as GHC generics. This allows us to define algorithms that work on arbitrary data structures.

Type Quantifiers

Consider the type of `fst`:

```
fst :: (a,b) -> a
```

This can be written more verbosely as:

```
fst :: forall a b. (a,b) -> a
```

Or, in a more mathematical notation:

$$\mathsf{fst} :: \forall a\ b.\ (a,b) \rightarrow a$$

This kind of quantification over type variables is called parametric polymorphism or just polymorphism for short.
(It's also called generics in some languages, but this terminology is bad)

What is the analogue of $\forall$ in logic? (via Curry-Howard)?

Curry-Howard

The type quantifier $\forall$ corresponds to a universal quantifier $\forall$, but it is not the same as the $\forall$ from first-order logic. What's the difference?

First-order logic quantifiers range over a set of *individuals* or values, for example the natural numbers:

$$\forall x.\ x + 1 > x$$

These quantifiers range over propositions (types) themselves. It is analogous to *second-order logic*, not first-order:

$$\forall A.\ \forall B.\ A \wedge B \to B \wedge A$$
$$\forall A.\ \forall B.\ (A, B) \to (B, A)$$

The first-order quantifier has a type-theoretic analogue too (type indices), but this is not nearly as common as polymorphism.

Generality

If we need a function of type $\texttt{Int} \to \texttt{Int}$, a polymorphic function of type $\forall a.\ a \to a$ will do just fine, we can just instantiate the type variable to $\texttt{Int}$. But the reverse is not true. This gives rise to an ordering.

Generality

A type A is *more general* than a type B, often written $A \sqsubseteq B$, if type variables in A can be instantiated to give the type B.

Example (Functions)

$$\texttt{Int} \to \texttt{Int} \quad \sqsupseteq \quad \forall z.\ z \to z \quad \sqsupseteq \quad \forall x\ y.\ x \to y \quad \sqsupseteq \quad \forall a.\ a$$

Constraining Implementations

How many possible total, terminating implementations are there of a function of the following type?

$$\texttt{Int} \rightarrow \texttt{Int}$$

How about this type?

$$\forall a.\ a \rightarrow a$$

Polymorphic type signatures constrain implementations.

Parametricity

Definition

The principle of parametricity states that the result of polymorphic functions cannot depend on values of an abstracted type.
More formally, suppose I have a polymorphic function g that is polymorphic on type a. If run any arbitrary function $f :: a \to a$ on all the a values in the input of g, that will give the same results as running g first, then f on all the a values of the output.

Example

$$foo :: \forall a.\ [a] \to [a]$$

We know that **every** element of the output occurs in the input.
The parametricity theorem we get is, for all f:

$$foo \circ (map\ f) = (map\ f) \circ foo$$

More Examples

$$head :: \forall a.\ [a] \to a$$

What's the parametricity theorems?

Example (Answer)

For any f:

$$f\ (head\ \ell) = head\ (map\ f\ \ell)$$

More Examples

$$(+\!\!+) :: \forall a.\ [a] \rightarrow [a] \rightarrow [a]$$

What's the parametricity theorem?

Example (Answer)

$$map\ f\ (a +\!\!+ b) = map\ f\ a +\!\!+ map\ f\ b$$

More Examples

$$concat :: \forall a.\ [[a]] \rightarrow [a]$$

What's the parametricity theorem?

Example (Answer)

$$map\ f\ (concat\ ls) = concat\ (map\ (map\ f)\ ls)$$

Higher Order Functions

$$\mathit{filter} :: \forall a.\ (a \to \mathit{Bool}) \ \to [a] \to [a]$$

What's the parametricity theorem?

Example (Answer)

$$\mathit{filter}\ \ p\ (\mathit{map}\ f\ \ \mathit{ls}) = \mathit{map}\ f\ (\mathit{filter}\ (p \circ f)\ \mathit{ls})$$

Parametricity Theorems

Follow a similar structure. In fact it can be mechanically derived, using the *relational parametricity* framework invented by John C. Reynolds, and popularised by Wadler in the famous paper, "Theorems for Free!"[1].
Upshot: We can ask `lambdabot` on the Haskell IRC channel for these theorems.

[1] `https://people.mpi-sws.org/~dreyer/tor/papers/wadler.pdf`

That's it

We have now covered all the content in COMP3141. Thanks for sticking with the course.

- **Functional Programming**
  - Basics: Introduction to Haskell, Algebraic Data Types, Type Classes
  - Advanced Programming: Higher-Kinded Abstractions, Controlling Effects
  - Advanced Types: Rich Types and GADTs, Polymorphism, Parametricity, Curry-Howard
- **Devising, Testing and Proving Program Properties**
  - Equational Reasoning and Inductive Proofs,
  - Property-Based Testing using QuickCheck,
  - Program Specifications: Data Invariants, Data Refinement, and Functional Correctness.

Reminders

- There is a quiz for this week, but no exercise (there's still Assignment 2).
- Curtis is giving a **revision lecture** this Thursday 3pm.
- Next week's lectures consist of an **invited lecture** and a second **revision lecture** on Friday with Curtis.
- Please come up with **questions** to ask Curtis for the revision lecture! It will be over very quickly otherwise.

Further Learning

- UNSW courses:
  - COMP3161 — Concepts of Programming Languages
  - COMP4161 — Advanced Topics in Verification
  - COMP6721 — (In-)formal Methods
  - COMP3131 — Compilers
  - COMP4141 — Theory of Computation
  - COMP6752 — Modelling Concurrent Systems
  - COMP3151 — Foundations of Concurrency
  - COMP3153 — Algorithmic Verification
- Online Learning
  - Oregon Programming Languages Summer School Lectures (https://www.cs.uoregon.edu/research/summerschool/archives.html) Videos are available from here! Also some on YouTube.