COMP3258

Functional Programming

Tutorial Session 2: Types, Curried Functions and QuickCheck

Overview

- Types
- Curried Functions
- QuickCheck (Optional)

Types

- A type is a name for a collection of related values.
- In GHCi, use :t or :type to infer the type of a given expression.

Types

```
ghci> :t not False
```

```
ghci> :t ['a', 'b', 'c']
```

```
ghci> :t [init, tail, reverse]
```

```
ghci> :t (False, True)
```

```
ghci> :t not
```

Type Inference

- Haskell supports definitions with or without a type declaration;
- When a definition does not have a type declaration
  - Haskell would automatically infers a type for the definition
  - and emit an error when it's not able to do so.

Parametric Polymorphism

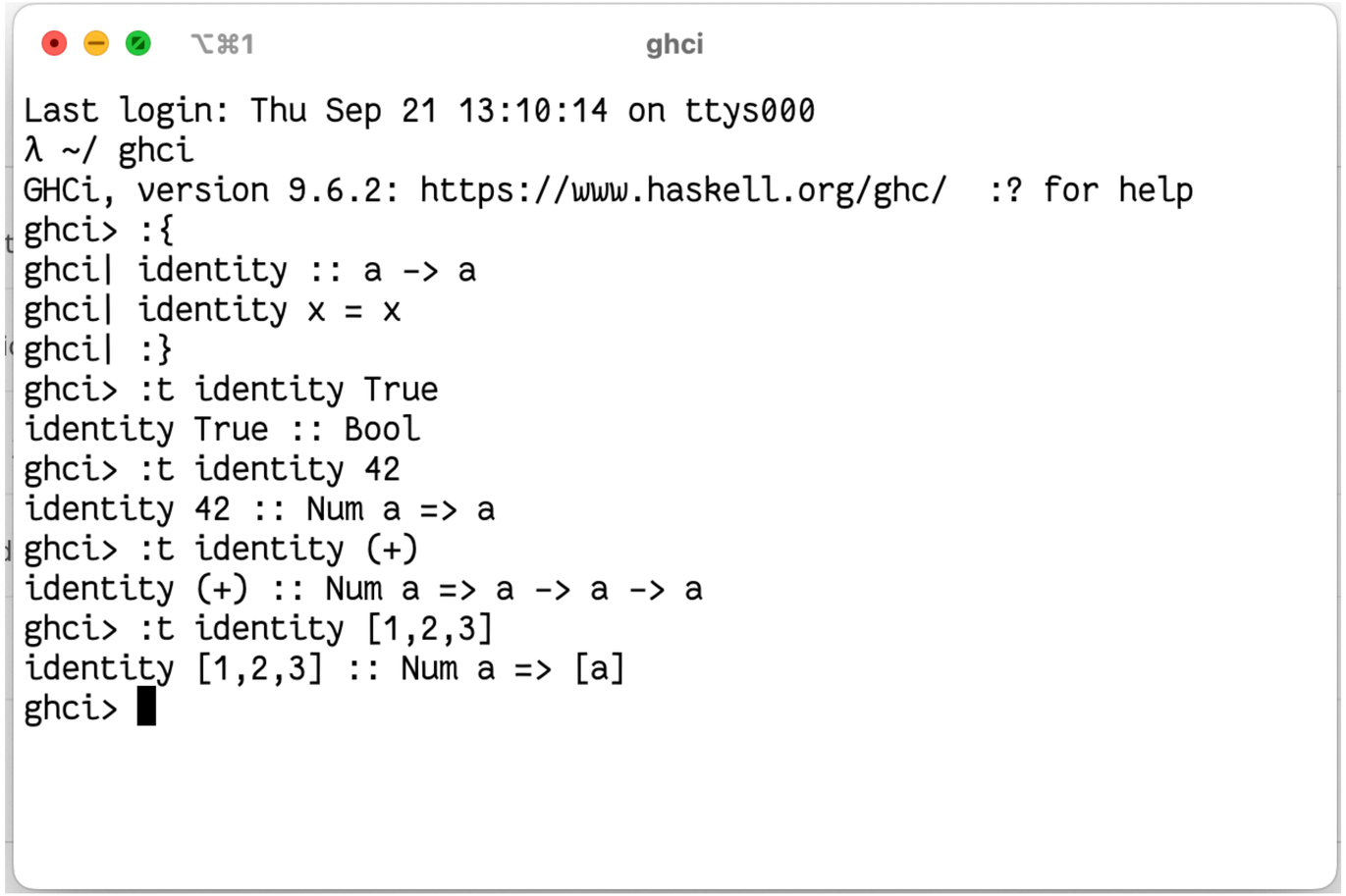
- Certain definitions may work for different types of parameters or return value.
  - consider **identity x = x**

`identity :: a → a`

- The identity function takes an arbitrary argument and returns that argument itself.
- a is a type variable (or type parameter)
  - in which an arbitrary type can fit.

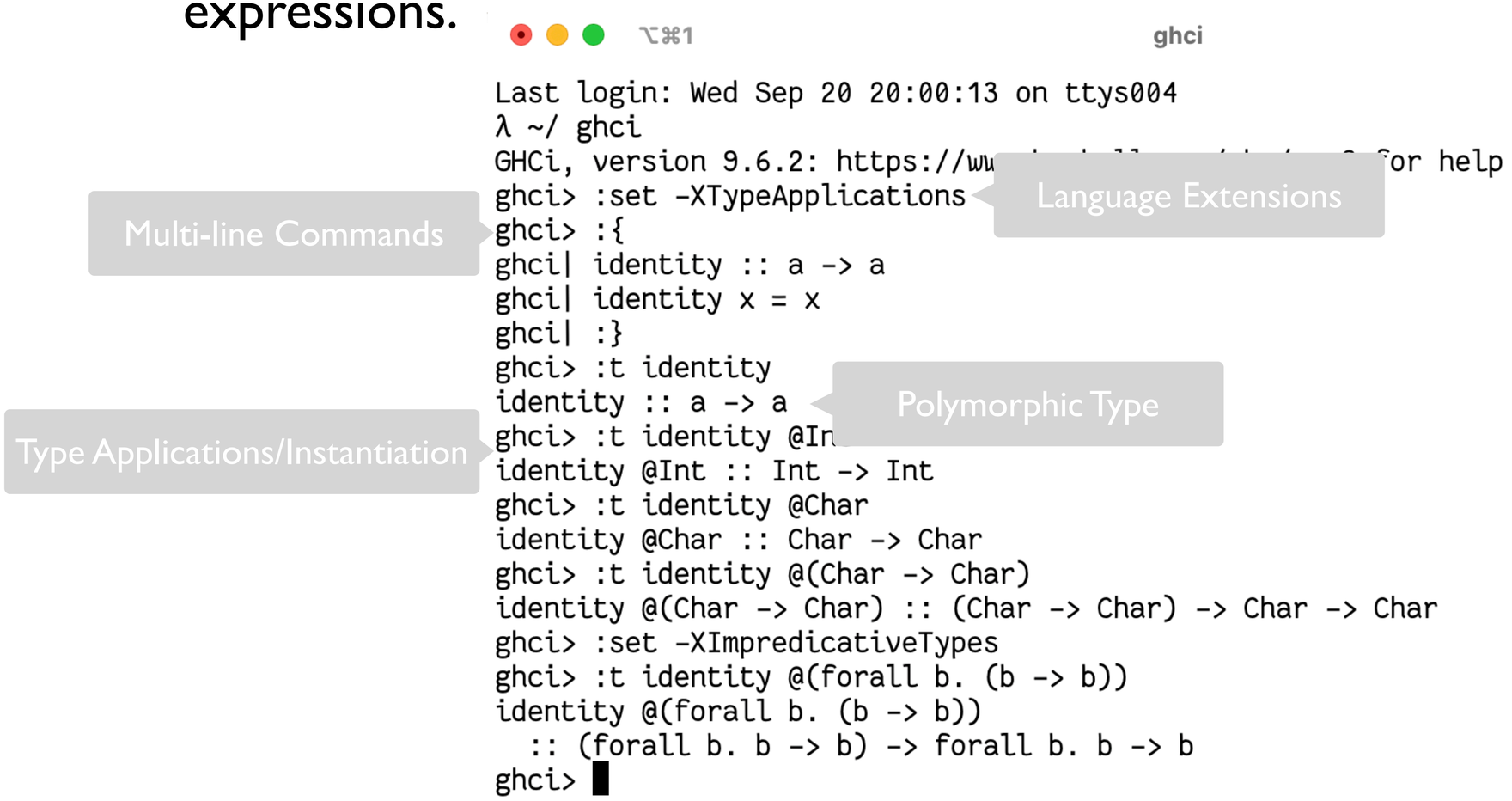
Function Application: Type instantiation (Implicit)

- Function application (function call) in Haskell will implicitly instantiate a polymorphic type.



TypeApplications: Type instantiation (Explicit)

- The TypeApplications is a extension allows you to use visible type application in expressions.



Curried Functions

- Functions with multiple arguments are also possible by returning functions as result.
- In Haskell, multi-argument functions are default curried.
  - Try: `listConcat xs ys = xs ++ ys`

Curried vs. Uncurried

```haskell
addCurried :: Int → Int → Int
addCurried x y = x + y

add1Curried :: Int → Int
add1Curried = addCurried 1
```

```haskell
addUncurried :: (Int, Int) → Int
addUncurried (x, y) = x + y

add1Uncurried :: Int → Int
add1Uncurried x = addUncurried (x, 1)
```

Question 1

- In Haskell Prelude library, there are two functions curry that converts an un-curried function to a curried one, and uncurry vice versa with signatures:

```
curry   :: ((a, b) → c) → (a → b → c)
uncurry :: (a → b → c) → ((a, b) → c)
```

```
curry :: ((a, b) → c) → a → b → c
curry f a b = f (a, b)

uncurry :: (a → b → c) → (a, b) → c
uncurry f (a, b) = f a b
```

Conditionals, Guards and Patterns

```
signum :: Int → Int
signum n = if n < 0 then -1 else
    if n == 0 then 0 else 1
```

```
signum :: Int → Int
signum n | n < 0     = -1
         | n == 0    = 0
         | otherwise = 1
```

```
not :: Bool → Bool
not False = True
not True = False
```

Question 2

- Pattern matching can not only be used with lists but also on various different types. Define two functions using pattern matching:
  - first that takes a pair as an argument, and returns the first element of the pair. (Hint: use pattern matching on pairs)
  - isZero that takes an integer as an argument and checks whether the integer is 0 or not. (Hint: use pattern matching on integers)

```
first :: (a, b) → a
first (a, _) = a

isZero :: Int → Bool
isZero 0 = True
isZero _ = False
```

Question 3

- Define a function safetail that behaves in the same way as tail, except that safetail maps the empty list to the empty list, whereas tail gives an error in this case.
- Define safetail using:
  - conditional expression
  - guarded equation
  - pattern matching

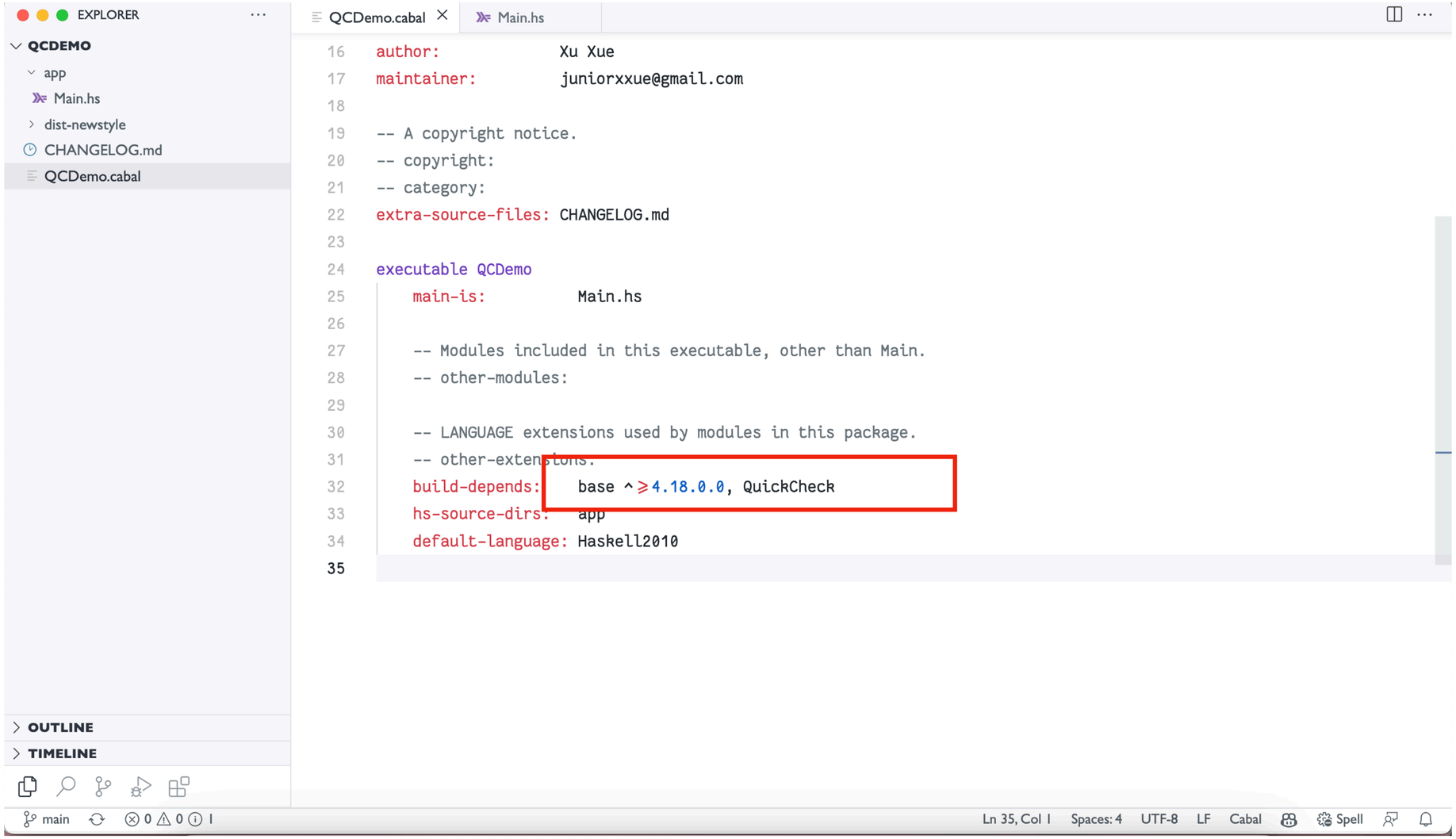
```haskell
-- if .. then .. else
safeTail :: [a] → [a]
safeTail xs = if null xs then [] else tail xs

-- guard
safeTail :: [a] → [a]
safeTail xs
  | null xs = []
  | otherwise = tail xs

-- pattern matching (recommended in practice)
safeTail :: [a] → [a]
safeTail [] = []
safeTail (x : xs) = xs
```

QuickCheck: Property Testing*

- It's a property testing package of Haskell.
- Properties are described as functions.
- Functions are automatically tested on random inputs.

Further reading: "QuickCheck: a lightweight tool for random testing of Haskell programs"

Install & Import QuickCheck



Specify Properties (Laws)

```
prop_RevUnit :: Int → Bool
prop_RevUnit x =
    reverse [x] == [x]

prop_RevApp :: [Int] → [Int] → Bool
prop_RevApp xs ys =
    reverse (xs ++ ys) == reverse ys ++ reverse xs

prop_RevRev :: [Int] → Bool
prop_RevRev xs =
    reverse (reverse xs) == xs
```

Specify Properties (Laws)

- The programmer must specify a fixed type at which the law is to be tested.
  - a fixed type simply means non-polymorphic
- we can use *parametricity* to argue that a property holds polymorphically.

Conditional Properties (Implication)

```
prop_MaxLe :: Int → Int → Property
prop_MaxLe x y =
    x ⩽ y ⟹ max x y == y

ordered :: [Int] → Bool
ordered [] = True
ordered (x:xs) = all (⩾ x) xs && ordered xs

insert :: Int → [Int] → [Int]
insert x [] = [x]
insert x (y:ys) = if x ⩽ y then x : y : ys else y : insert x ys

prop_Insert :: Int → [Int] → Property
prop_Insert x xs =
    ordered xs ⟹ ordered (insert x xs)
```

Conditional Properties (Implication)

- Use implication (==>) to express conditional properties;
- The type of property “Bool” are replaced by “Property”.

Monitor Random Tests

- use classify to separate out trivial cases;
- use collect to list test cases according to certain measures.

Monitor Random Tests

```
prop_InsertClassify :: Int → [Int] → Property
prop_InsertClassify x xs =
    ordered xs ⟹
        classify (null xs) "trivial cases" $
            ordered (insert x xs)

prop_InsertCollect :: Int → [Int] → Property
prop_InsertCollect x xs =
    ordered xs ⟹
        collect (length xs) $
            ordered (insert x xs)
```

Monitor Random Tests

QCDemo.cabal | Main.hs

```
64  main :: IO ()
65  main = do
66      -- quickCheck prop_RevUnit
67      -- quickCheck prop_RevApp
68      -- quickCheck prop_RevRev
69      -- quickCheck prop_MaxLe
70      -- quickCheck prop_Insert
71      quickCheck prop_InsertClassify
72      quickCheck prop_InsertCollect
73      -- quickCheck prop_InsertOrdered
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL

```
λ ~/Library/CloudStorage/Dropbox/cs/2023comp3258/tutorials/02/QCDemo/ cabal run
Up to date
*** Gave up! Passed only 72 tests; 1000 discarded tests (26% trivial cases).
*** Gave up! Passed only 65 tests; 1000 discarded tests:
38% 0
34% 1
12% 2
11% 3
 5% 4
λ ~/Library/CloudStorage/Dropbox/cs/2023comp3258/tutorials/02/QCDemo/
```

Custom Test Data Generator

```
prop_InsertOrdered :: Int → Property
prop_InsertOrdered x =
    forAll orderedList $ \xs →
        ordered (insert x xs)
```

QuickCheck Primitives

```
import Test.QuickCheck
    ( orderedList,
      (⟹),
      classify,
      collect,
      forAll,
      quickCheck,
      Property )
```

Reminder

- The first assignment will be released next week (Perhaps Monday)
- Next tutorial will cover recursive functions and sorting algorithms.