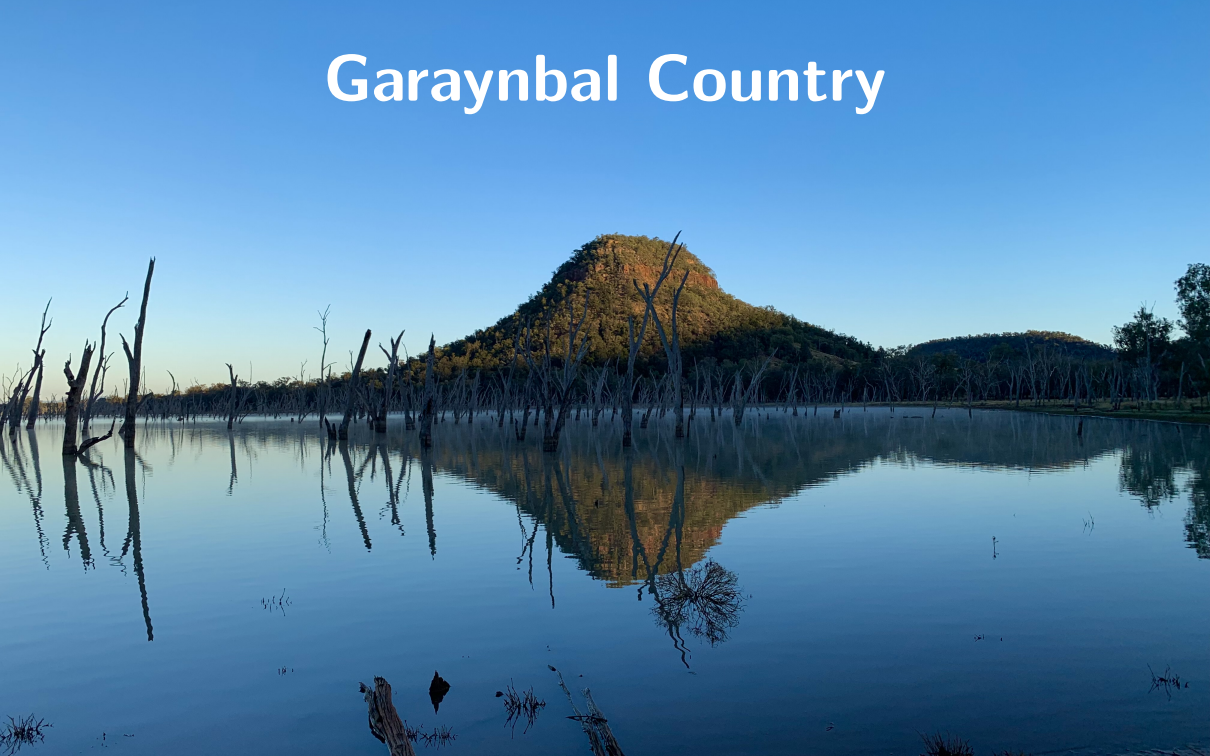
Garaynbal Country



We acknowledge, celebrate and pay our respects to the Ngunnawal and Ngambri people of the Canberra region and to all First Nations Australians on whose traditional lands we meet and work, and whose cultures are among the oldest continuing cultures in human history.

Learn more about Acknowledgement of Country here

Find out more about Canberra's Aboriginal history here

Parametric polymorphism

Contents

Lists

(Thompson: §5.4 and §5.5)

Remember lists

```
1  data [Int] = [] | Int : [Int]
2  data [Char] = [] | Char : [Char]
3  data [Int -> Bool] = [] | (Int -> Bool) : [Int -> Bool]
```

In general

```
4  data [a] = [] | a : [a]
```

- This is a **recursive type**, where `[a]` is defined by referring to itself
- It is a **polymorphic data type**, because we can replace `a` for any other type
- `a` is called a **type variable**

Maybe

(Thompson: §14.7)

The `Maybe` type is defined as

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

This is a **polymorphic data type**, because we can replace the type variable `a` for any type

Example If `a` is `String` we get

```
2  data Maybe String = Nothing | Just String
```

Remark The `Maybe` type can be used if we are not sure if we have an element of type `a` or not. We will see it in the labs

Pairs

(Thompson: §5.1 and §5.2)

Pairs are defined by

```
1  data (,) a b = (,) a b
```

- We usually write `(a,b)` instead of `(,) a b`
- `a` and `b` are type variables

Example We can define a polymorphic function `fst`

```
2  fst :: (a, b) -> a
3  fst (x,_) = x
```

```
ghci> fst ("hello", 123)
"hello"
```

```
ghci> fst ("hello","world")
"hello"
```

Tuples

(Thompson: §5.1 and §5.2)

The construction `(,)` can be used for more than two elements

```
1  data (,) a b = (,) a b
2  data (,,) a b c = (,,) a b c
3  data (,,,) a b c d = (,,,) a b c d
4  data (,,,,) a b c d e = (,,,,) a b c d e
5  data (,,,,,) a b c d e f = (,,,,,) a b c d e f
```

Datatypes with variables

(Thompson: §6.1)

A **polymorphic datatype** is a type that uses type variables

- Make a language more expressive, while maintaining full **static type-safety**
  - every Haskell expression has a type
  - types are all checked at compile-time
  - programs with type errors will not compile
- **Type variables** stand for arbitrary types
  - Easy to spot because they start with lower case letters
  - Usually we just use one letter on its own, e.g. `a`, `b`, `c`
- Sometimes called **generic types**

Polymorphisms in Haskell

(Thompson: §6.1)

A **polymorphism** is a function between types that have type variables

- Two kinds of polymorphism:
  - **parametric**
  - **ad hoc** (coming later!)
- Using parametric polymorphism, a function can be written **generically** so it can handle values **without depending on their type**
- Replacing type variables with specific types is called **instantiation**, the resulting function is an **instance** of the polymorphic function

Identity function

```
1  id :: a -> a
2  id x = x
```

Lists

Remember lists

```
1  data [a] = [] | a : [a]
```

Remarks

- `[a]` is sometimes written as `[] a`
- There is lots of syntactic sugar to make list programming more pleasant

When to use parametric polymorphisms?

To decide if a list function should be parametric polymorphic, simply ask:

"Does my function depend on the type of elements in the list?"

Some parametric polymorphic list functions

(Thompson: §6.2)

function	type	description	example
`:`	`a -> [a] -> [a]`	add a single element to the front:	`3:[2,4] = [3,2,4]`
`++`	`[a] -> [a] -> [a]`	join two lists together:	`"Has" ++ "kell"= "Haskell"`
`!!`	`[a] -> Int -> a`	return *n*-th element, counting from 0:	`[2,4,7] !! 1 = 4`
`concat`	`[[a]] -> [a]`	concatenate a list of lists:	`concat [[2,3],[],[4]] = [2,3,4]`
`length`	`[a] -> Int`	give length of a list:	`length "word"= 4`
`head,last`	`[a] -> a`	give first/last element of a list:	`head "word"= w`
`tail,init`	`[a] -> [a]`	all except for first/last element:	`tail "word"= "ord"`
`replicate`	`Int -> a -> [a]`	repeat an element *n* times	`replicate 3 'a' = "aaa"`
`take`	`Int -> [a] -> [a]`	take *n* elements from the front	`take 3 "Haskell"= "Has"`
`drop`	`Int -> [a] -> [a]`	drop *n* elements from the front	`drop 3 "Haskell"= kell`
`splitAt`	`Int -> [a] -> ([a],[a])`	split at a given position	`splitAt 3 "Haskell"= ("Has","kell")`
`reverse`	`[a] -> [a]`	reverse order of elements	`reverse "hey!" = "!yeh"`
`zip`	`[a] -> [b] -> [(a,b)]`	list of pairs of elements	`zip [1,2] [3,4,5] = [(1,3),(2,4)]`
`unzip`	`[(a,b)] -> ([a],[b])`	turn a list of pairs into a pair of lists	`unzip [(1,5),(2,6)] = ([1,2],[5,6])`

Concat

(Thompson: §6.2)

The function `concat` glues together a list of lists into one long list

```
1  concat :: [[a]] -> [a]
```

Example

```
2  ghci> concat [[1,2,3],[4,5],[],[6,7,8]]
3  [1,2,3,4,5,6,7,8]
4
5  ghci> concat ["hello"," ","world"]
6  "hello world"
7
8  ghci> concat [[1,2,3],[False,True]]
```

Some monomorphic list functions

The functions

```
and :: [Bool] -> Bool
or :: [Bool] -> Bool
```

take the conjunction and disjunction of a list of Boolean values.

Example

```
ghci> and [True, True, False]
False

ghci> or [True, True, False]
True
```

Remark These functions are **monomorphic**: we cannot replace `Bool` with an arbitrary type

List functions in the prelude

Warning The types of some `Prelude` list functions look like

```
ghci> :t length
length :: [a] -> Int
```

but they actually are

```
ghci> :t length
length :: Foldable t => t a -> Int
```

For now, when we see `Foldable t` we think of each `t a` as `[a]`

Summary

- A **polymorphic datatype** is a type that uses type variables
- A **polymorphism** is a function between types that have type variables
- Filling in the type variables for concrete types is called **instantiation**, the resulting type or function is called an **instance**
- There are many (built-in) polymorphisms for lists
- A **monomorphism** is a function that is not a polymorphism

Next

Labs

- Week 5: recursion with lists
- Week 6: parametric polymorphisms

Lectures

- Wednesday: Code quality
- Wednesday: Good report writing
- Thursday: Example solutions for midsemester quiz