Session 4: Lists and Polymorphism

COMP2221: Functional programming

Laura Morgenstern

laura.morgenstern@durham.ac.uk

Recap

- Learned that functions have types
- Discussed currying as a manner to define functions with multiple arguments
- Introduced the idea of anonymous functions
- Saw syntax for these λ expressions in Haskell
- And how they can formalise (or make it easier to read) curried functions:

```haskell
add x y = x + y
-- vs
add = \x -> (\y -> x + y)
```

- Considered infix and prefix notation

Lists: pattern matching

Representation of lists

- Every non-empty list is created by repeated use of the `(:)` operator "*cons*truct" that adds an element to the start of a list

```
[1, 2, 3, 4] = 1 : (2 : (3 : (4 : [])))
```

- This is a representation of a *linked list*
- Operations on lists such as indexing, or computing the length must therefore *traverse* the list.

⇒ Operations such `reverse`, `length`, `(!!)` are linear in the length of the list.

- Getting the `head` and `tail` is constant time, as is `(:)` itself.

Pattern matching on lists

- lists can be used for pattern matching in function definitions

```
startsWithA :: [Char] -> Bool
startsWithA ['a', _, _] = True
startsWithA _ = False
```

- Matches 3-element lists and checks if the first entry is the character `'a'`.

Careful

Use patterns in the equations defining a function. Not in the type of the function.

Pattern matches in the equations don't change the *type* of the function. They just say how it should act on particular expressions.

Pattern matching on lists

- How match `'a'` and not care how long the list is?
- Can't use literal list syntax. Instead, use list constructor syntax for matching.

```
startsWithA :: [Char] -> Bool
startsWithA ('a':_) = True
startsWithA _ = False
```

- `('a':_)` matches any list of length *at least* 1 whose first entry is `'a'`.
- The *wildcard* match `_` matches anything else.
- This works to match multiple entries too:

```
startsWithAB :: [Char] -> Bool
startsWithAB ('a':'b':_) = True
startsWithAB _ = False
```

Binding variables in pattern matching

- As well as matching literal values, we can also match a (list) pattern, and bind the values.

```
sumTwo :: Num a => [a] -> a
sumTwo (x:y:_) = x + y
```

- Match lists of length *at least* two and sum their first two entries

Example

```
sumTwo [1, 2, 3, 4]
-- introduces the bindings
x = 1
y = 2
_ = [3, 4]
```

- Reminder: can't repeat variable names in bindings (exception _)

```
-- Not allowed
sumThree (a:a:b:_) = a + a + b
-- Allowed
second (_:a:_) = a
```

What types of pattern can I match on?

- Patterns are constructed in the same way that we would construct the arguments to the function

```
(&&) :: Bool -> Bool -> Bool
True && True = True
False && _ = False
-- Used as:
a && b
```

```
head :: [a] -> a
head (x:_) = x
-- Used as:
head [1, 2, 3] == head (1:[2, 3])
```

- This is a general rule in constructing pattern matches "If I were to call the function, what structure do I want to match?"
- Caveat: can only match "data constructors"

```
-- Not allowed
last :: [a] -> a
last (xs ++ [x]) = x
```

Lists: comprehensions

List comprehensions I: syntax

- In maths, we often use *comprehensions* to construct new *sets* from already defined ones

$$\{2, 4\} = \{x \mid x \in \{1..5\}, x \bmod 2 = 0\}$$

"The set of all integers x between 1 and 5 such that x is even."

- Haskell supports similar notation for constructing lists.

```
Prelude> [x | x <- [1..5], x `mod` 2 == 0]
[2, 4]
```

"The list of all integers x where x is drawn from `[1..5]` and x is even"

- `x <- [1..5]` is called a *generator*
- Compare Python comprehensions

```
[x for x in range(1, 6) if (x % 2) == 0]
```

List comprehensions II: generators

- Comprehensions can contain multiple generators, separated by commas

```
Prelude> [(x, y) | x <- [1,2,3], y <- [4, 5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

- Variables in the later generator change faster: analogous to nested loops

```
l = []
for x in [1, 2, 3]:
  for y in [4, 5]:
    l.append((x, y))

# analogously
[(x, y) for x in [1, 2, 3] for y in [4, 5]]
```

- Later generators can reference variables from earlier generators

```
Prelude> [(x, y) | x <- [1..3], y <- [x..3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

"All pairs (x, y) such that $x, y \in \{1, 2, 3\}$ and $y \geq x$"

List comprehensions III: guards

- As well as binding variables to values with generators, we can restrict the values using *guards*
- A guard can be any function that returns a `Bool`
- Guards and generators can be freely interspersed, but guards can only refer to variables to their left

```
Prelude> [(x, y) | x <- [1..3], even x, y <- [x..3]]
[(2, 2), (2, 3)]
Prelude> [(x, y) | x <- [1..3], y <- [x..3], even x, even y]
[(2, 2)]
Prelude> [(x, y) | x <- [1..3], even x, even y, y <- [x..3]]
error: Variable not in scope: y :: Integer
```

Some examples

- Produce a list of all factors of some positive integer

```
factors :: Int -> [Int]
factors n = [x | x <- [1..n], n `mod` x == 0]
```

- For example

```
> factors 10
[1, 2, 5, 10]
```

- Now we can determine if a number is prime

```
prime :: Int -> Bool
prime n = factors n == [1, n]
```

- And use it to (very expensively) enumerate primes below a limit

```
primes :: Int -> [Int]
primes n = [x | x <- [2..n], prime x]
```

Polymorphism

Polymorphism

- Recall, Haskell is *strictly typed*.
- What does this mean for (say) `length`?

Different types?

```
length [True, False, True] -- :: [Bool] -> Int ?
length [1, 2, 3]           -- :: [Int] -> Int ?
```

These functions must have *different* types, no?

Polymorphism

- Recall, Haskell is *strictly typed*.
- What does this mean for (say) `length`?

Different types?

```
length [True, False, True] -- :: [Bool] -> Int ?
length [1, 2, 3]           -- :: [Int] -> Int ?
```

These functions must have *different* types, no?

Polymorphic types

```
Prelude> :type length
length :: [a] -> Int
```

"`length` eats a list of values of any type a and returns an Int"

a is called a *type variable*.

This is called *parametric polymorphism*.

Contrast with OO languages: defintions

Definition (Parametric polymorphism)

Write a *single* implementation of a function that applies generically *and identically* to values of any type.

Definition ("ad-hoc" polymorphism)

Write *multiple* implementations of a function, one for each type you wish to support.

Definition (Subtype polymorphism)

Relate datatypes by some "substitutability". Write a function for a supertype instance. Now all subtypes can use it. (see also "Liskov substitution principle")

Contrast with OO languages: examples

Subtype polymorphism

```python
class Foo(object):
   def length(self, ...):
     pass
class Bar(Foo):
   pass
a = Foo().length()
# Every Bar is-a Foo, so we can
# call the length method.
b = Bar().length()
```

Ad-hoc polymorphism

```python
class Foo(object):
   pass
class Bar(object):
   pass
def length(obj):
   if isinstance(obj, Foo):
      ...
   elif isinstance(obj, Bar):
      ...
# length knows how to handle things
# of type Foo and type Bar
a = length(Foo())
b = length(Bar())
```

Parametric polymorphism

```haskell
-- length doesn't care what type the entries
-- in the list are
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

Contrast with OO languages

- Parametric polymorphism also called *generic programming*
- Introduced in ML in 1975.
- Has been adopted by a number of languages, including traditional OO ones.
- For example, Java or C# have "generics" for this purpose

```java
// Implementation of HashSet is generic
// Specialised on instantiation
Set<Object> objset = new HashSet<Object>();
```

- C++ templates also allow for similar style of programming

Constraining polymorphic functions

- Some polymorphic functions only apply to types that satisfy certain constraints
- For example `(+)` works on all types `a`, *as long as* that type is a number type.

Example

```
(+) :: Num a => a -> a -> a
```

"For any type `a` that is an *instance* of the *class* `Num` of numeric types, `(+)` has type `a -> a -> a`"

- This constraint is called a *class constraint*
- An expression or type with one or more such constraints is called *overloaded*.

⇒ `Num a => a -> a -> a` is an *overloaded type* and `(+)` is an *overloaded function*.

Haskell classes

Definition (Class)

A collection of *types* that support certain, specified, overloaded operations called *methods*.

Definition (Instance)

A concrete type that belongs to a *class* and provides implementations of the required methods.

Analogous constructs in other languages

- Compare: type "a collection of related values"
- This is *not* like subclassing and inheritance in Java/C++
- If you write flat interfaces with 'abc.abstractmethod' in Python.
- Rust traits give you something close
- Close to a combination of Java *interfaces* and *generics*
- C++ "concepts" (in C++20) are also very similar.

Defining classes I

- Let us say we want to encapsulate some new property of types `Foo`-ness
- We define the interface the type should support

```
class Foo a where
  isfoo :: a -> Bool
```

- Now we say how types implement this

```
instance Foo Int where
  isfoo _ = False

instance Foo Char where
  isfoo c = c `elem` ['a'..'c']
```

- Can add new interfaces to old types, and new types to old interfaces.
- Contrast Java, where if I implement a new interface it is very difficult to make existing classes implement it.

Defining classes II

- Classes (interfaces) can provide default implementation.
- Example, the `Eq` class representing equality requires both `(==)` and `(/=)`.
- Since `a == b` ⇔ `not (a /= b)`, we can provide *default* implementations and only require that an instance implements one.

```
class Eq a where
  (==) :: a -> a -> Bool
  x == y = not (x /= y)
  (/=) :: a -> a -> Bool
  x /= y = not (x == y)

-- instance for MyType only needs to provide one of (==) or (/=).
instance Eq MyType where
  x == y = ...
```

Summary

- Saw how the literal list syntax translates into construction with `(:)`
- Discussed complexity of common list operations
- Made connection to pattern matching of lists
- Introduced list comprehensions as analogous to set notation
- Saw how nested comprehensions and guards work
- Saw how Haskell implements *polymorphism* through generic functions

```haskell
-- length operates on a list of any type a
-- and returns an Int
length :: [a] -> Int
```

- Saw how overloading works with *class constraints* and *type classes*

```haskell
-- sort sorts any list of things of type a,
-- as long as that type is orderable
sort :: Ord a => [a] -> [a]
```