Session 5: Polymorphism and Custom Data Types

COMP2221: Functional programming

Laura Morgenstern

laura.morgenstern@durham.ac.uk

Recap

- 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
```

Recap: Types of Polymorphism

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

```
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

```
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

```
-- 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

WARNING!

The *words* class and instance are the same as in object-oriented programming languages, but their *meaning* is very different.

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 = ...
```

Adding new data types

Defining data types

- It often makes sense to *define* new data types
- Multiple reasons to do this:
  1. Hide complexity
  2. Build new abstractions
  3. Type safety
- Haskell has three ways to do this
  - `type`
  - `data`
  - `newtype` (we won't cover this one)

Type declarations: new names, old types

- A new *name* for an existing type can be defined using a *type declaration*

`String` as a synonym for the type `[Char]`

```haskell
type String = [Char]

vowels :: String -> [Char]
vowels str = [s | s <- str, s `elem` ['a', 'e', 'i', 'o', 'u']]

Prelude> vowels "word"
"o"
Prelude> vowels ['w', 'o', 'r', 'd']
"o"
```

- Notice that there is no type distinction: objects of type `String` and `[Char]` are completely interchangeable.

New names, old types II

- We can use these type declarations to make the semantics of our code clearer

An integer position in 2D

```
type Pos = (Int, Int)

origin :: Pos
origin = (0, 0)

left :: Pos -> Pos
left (i, j) = (i - 1, j)
```

- Reader has to expend less brain power to understand the function
- Similar to C's `typedef`

New names, old types III

- Just like function definitions, type declarations can be parameterised over *type variables*

Example

```
type Pair a = (a, a)

mult :: Pair Int -> Int
mult (m, n) = m*n

dup :: a -> Pair a
dup x = (x, x)
```

✗ Can't use *class constraints* in the definition

✗ Can't have *recursive* types

Not allowed

```
Prelude> type Tree = (Int, [Tree])
error:
    Cycle in type synonym declarations:
```

Data declarations: new types

- We can introduce a completely *new* type by specifying allowed values using a *data declaration*

A boolean type

```
data Bool = False | True
```

"Bool is a new type, with two new values: False, and True"

- The two values are called *constructors* for the type `Bool`
- Both the type name, and the constructor names, must begin with an upper-case letter.
- This is actually the way `Bool` is implemented in the standard library

Using new types

- Once defined, we can use new types exactly like built in ones

Example

```
data IsTrue = Yes | No | Perhaps

negate :: IsTrue -> IsTrue
-- Pattern matching on constructors
negate Yes     = No
negate No      = Yes
negate Perhaps = Perhaps

Prelude> negate Perhaps
Perhaps
```

Data declarations with fixed type parameters

- The constructors in a data declaration can take arbitrarily many parameters

Example

```
data Shape = Circle Float | Rectangle Float Float
```

"A shape is either a Circle, or a Rectangle. The Circle is defined by one number, the Rectangle by two"

Pattern matching on the constructors:

```
area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rectangle x y) = x * y
```

Data declarations with type variables

- We can also make our data declarations *polymorphic* with appropriate type variables

Example

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

"A `Maybe` is either `Nothing` or else a `Just` with a value of arbitrary type"

```
safehead :: [a] -> Maybe a
safehead [] = Nothing
safehead (x:_) = Just x
```

Recursive types

- Data declarations can *refer to themselves*

Peano numbers

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

"Nat is a new type with constructors `Zero :: Nat` and `Succ :: Nat -> Nat`"

- This type contains the infinite sequence of values

```
Zero
Succ Zero
Succ (Succ Zero)
...
```

- We could use this to implement a representation of the natural numbers, and arithmetic

```
add :: Nat -> Nat -> Nat
add Zero n = n
add (Succ m) n = Succ (add m n)
```

Recursive types II

- This kind of recursive type allows very succint definitions of data structures

Linked list

```
data List a = Empty | Cons a (List a)
intList = Cons 1 (Cons 2 (Cons 3 Empty))
== [1, 2, 3]
```

“A List is either Empty, or a Cons of a value and a List”

Linked list in C

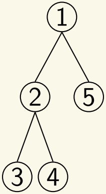
```
typedef struct _Link *Link;
struct _Link {
  void *data;
  Link next;
}
```

A binary tree

A binary tree with values at nodes

```
data BTree a = Empty | Node a (BTree a) (BTree a)
btree = Node 1 (Node 2 (Node 3 Empty Empty)
                       (Node 4 Empty Empty))
               (Node 5 Empty Empty)
```

"A BTree is either Empty, or a Node containing a value and two BTrees"



Pattern matching

- Recall the pattern matching syntax on lists

```
list = [1, 2, 3, 4] == 1:[2, 3, 4]
-- Binds tip to 1, rest to [2, 3, 4]
(tip:rest) = list
```

- The pattern matches the "constructor" of the list, as if the declaration were

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

- Exactly the same pattern matching applies to data types on their data constructors

```
data List a = Empty | Cons a (List a)
list = Cons 1 (Cons 2 (Cons 3 Empty))
-- Binds tip to 1, rest to (Cons 2 (Cons 3 Empty))
(Cons tip rest) = list
```

Some type theory and contrasts

- Haskell's `data` declarations make *Algebraic data types*
- This is a type where we specify the "shape" of each element
- The two algebraic operations are "sum" and "product"

Definition (Sum type)

An alternation:

```
data Foo = A | B
```

A value of type `Foo` can either be `A` or `B`

Definition (Product type)

A combination:

```
data Pair = P Int Double
```

a pair of numbers, an `Int` and `Double` together.

Other languages: product types

- Almost all languages have *product types*. They're just "ordered bags" of things.
- In Python, we can use tuples or classes

Python

```python
pair = (1, 2)
x, y = pair
```

- In C we use structs

C struct

```c
struct Pair {                  struct Pair p;
  int x;                       p.x = 1;
  int y;                       p.y = 2;
}
```

- In Java, classes

Other languages: sum types

- Useful for type safety/compiler warnings: easy to statically prove that every option is handled
- Less common, although new languages are catching on (e.g. Rust, Swift)
- In C and Java for integers, you can use an `enum`

```
enum Weekdays {
 MON, TUE, WED, THU, FRI, SAT, SUN
};
```

Haskell types: pros and cons

Classes

- ✓ Easy to add new “kinds of things”: just make a subclass
- ✗ Hard to add new “operation on existing things”: need to change superclass to add new method and potentially update all subclasses

Algebraic data types

- ✗ Hard to add new “kinds of things”: need to add new constructor and update all functions that use the data type
- ✓ Easy to add new “operation on existing things”: just write a new function

Pros and Cons II

Adding new things

Just implement a new subclass

```
class Car(object):
    def seats(self): return 4
class MX5(Car):
    def seats(self): return 2
# Later
class Mini(Car): pass
```

Have to update data constructor

```
data Car = MX5
-- Later
data Car = MX5 | Mini
```

Adding new operations

Must update all classes

```
class Car(object):
    def mpg(self): return 25
    def seats(self): return 4
class MX5(Car):
    def mpg(self): return 30
    def seats(self): return 2
class Mini(Car):
    def mpg(self): return 40
```

Just write new functions

```
seats :: Car -> Int
seats MX5 = 2
seats Mini = 4
mpg :: Car -> Int
mpg MX5 = 30
mpg Mini = 40
```

Summary

- Saw how to define new types in Haskell
- Introduced `type` keyword for synonyms
- Introduced `data` for completely new types, and the introduction of data constructors
- Saw pattern matching for data constructors
- Contrasted sum and product types, and availability in other languages