Session 6: Custom Data Types and Recursion

COMP2221: Functional programming

Laura Morgenstern

laura.morgenstern@durham.ac.uk

Recap

- 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
- Considered recursive data types
- Saw pattern matching for data constructors

Type Theory

- 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
};
```

OO Classes vs. Algebraic Data Types: Adding new subtypes

OO Classes

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
```

Algebraic Data Types

Have to update data constructor (and hence all functions that use this type!)

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

Classes vs. Algebraic Data Types: Adding new operations

Classes

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
```

Algebraic Data Types

Just write new functions

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

Classes vs. Algebraic Data Types

Classes

- ✓ Easy to add new subtypes: just make a subclass
- ✗ Hard to add new operations on existing types: need to change superclass to add new method and potentially update all subclasses

Algebraic data types

- ✗ Hard to add new subtypes: need to add new constructor and update all functions that use the data type
- ✓ Easy to add new operations on existing types: just write a new function

Recursion

Recursion

Definition

recursion *noun*

see: recursion.

Definition

Recursion means to define something in terms of itself.

Advice when writing recursive functions

1. define the type
2. enumerate the cases
3. define the simple or *base* cases
4. define the reduction of other cases to simpler ones
5. (optional) generalise and simplify

Example: `drop`

1. define the type

Drop the first *n* elements from a list

```
drop :: Int -> [a] -> [a]
```

2. enumerate the cases
3. define the simple or *base* cases
4. define the reduction of other cases to simpler ones
5. (optional) generalise and simplify

Example: `drop`

1. define the type

Drop the first *n* elements from a list

```haskell
drop :: Int -> [a] -> [a]
```

2. enumerate the cases

Two cases each for the integer and the list argument

```haskell
drop 0 [] =
drop 0 (x:xs) =
drop n [] =
drop n (x:xs) =
```

3. define the simple or *base* cases
4. define the reduction of other cases to simpler ones
5. (optional) generalise and simplify

Example: `drop`

1. define the type
2. enumerate the cases

Two cases each for the integer and the list argument

```
drop 0 [] =
drop 0 (x:xs) =
drop n [] =
drop n (x:xs) =
```

3. define the simple or *base* cases

Zero and the empty list are fixed points

```
drop 0 [] = []
drop 0 (x:xs) = x:xs
drop n [] = []
drop n (x:xs) =
```

4. define the reduction of other cases to simpler ones
5. (optional) generalise and simplify

Example: `drop`

1. define the type
2. enumerate the cases
3. define the simple or *base* cases

Zero and the empty list are fixed points

```
drop 0 [] = []
drop 0 (x:xs) = x:xs
drop n [] = []
drop n (x:xs) =
```

4. define the reduction of other cases to simpler ones

Apply drop to the tail

```
drop 0 [] = []
drop 0 (x:xs) = x:xs
drop n [] = []
drop n (x:xs) = drop (n-1) xs
```

5. (optional) generalise and simplify

Example: `drop`

1. define the type
2. enumerate the cases
3. define the simple or *base* cases
4. define the reduction of other cases to simpler ones

Apply drop to the tail

```
drop 0 [] = []
drop 0 (x:xs) = x:xs
drop n [] = []
drop n (x:xs) = drop (n-1) xs
```

5. (optional) generalise and simplify

Compress cases

```
drop :: Int -> [a] -> [a]
drop 0 xs = xs
drop _ [] = []
drop n (x:xs) = drop (n-1) xs
```

Example: `drop`

1. define the type
2. enumerate the cases
3. define the simple or *base* cases
4. define the reduction of other cases to simpler ones
5. (optional) generalise and simplify

Compress cases

```
drop :: Int -> [a] -> [a]
drop 0 xs = xs
drop _ [] = []
drop n (x:xs) = drop (n-1) xs
```

6. And we're done (this is the standard library definition)

Equivalence of recursion and iteration

- Both purely iterative and purely recursive programming languages are Turing complete
- Hence, it is always possible to transform from one representation to the other
- Which is convenient depends on the algorithm, and the programming language

Recursion $\Rightarrow$ iteration

- Write looping constructs, manually manage function call stack

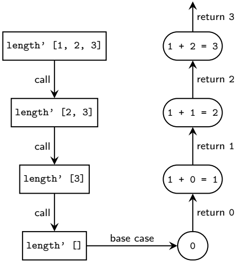
Iteration $\Rightarrow$ recursion

- Turn loop variables into additional function arguments
- and write a *tail recursive function* (see later)

How are function calls managed?

- Usually a *stack* is used to manage nested function calls

```
length :: [a] -> Int
length' [] = 0
length' (x:xs) = 1 + length' xs
Prelude> length' [1, 2, 3]
```



- Each entry on the stack uses memory
- Too many entries causes errors: the dreaded *stack overflow*
- How big this stack is depends on the language
- Typically "small" in imperative languages and "big" in functional ones

Typically don't have to worry about stack overflows

- In traditional *imperative* languages, we often try and avoid recursion
- Function calls are more expensive than just looping
- Deep recursion can result in stack overflow:

```
def fac(n): return 1 if n == 0 else n * fac(n-1)
> fac(3000)
RecursionError Traceback (most recent call last)
----> 1 def fac(n): return 1 if n == 0 else n * fac(n-1)
RecursionError: maximum recursion depth exceeded in comparison
```

- In contrast, Haskell is fine with much deeper recursion

```
fac n = if n == 0 then 1 else n * fac (n-1)
> fac(200000)
..... -- fine, if slow
```

- Unsurprising, given the programming model
- Still prefer to avoid recursion trees that are too deep

Classifying recursive functions I

- Since it is natural to write recursive functions, it makes sense to think about classifying the different types we can encounter
- Classifying the type of recursion is useful to allow us to think about better/cheaper implementations

Definition (Linear recursion)

The recursive call contains only a *single* self reference

```
length' []     = []
length' (_:xs) = 1 + length' xs
```

Function just calls itself repeatedly until it hits the base case.

Definition (Multiple recursion)

The recursive call contains *multiple* self references

```
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

Classifying recursive functions II

Definition (Direct recursion)

The function calls *itself* recursively

```
product' []     = []
product' (x:xs) = x * product' xs
```

Definition (Mutual/indirect recursion)

Multiple functions call *each other* recursively

```
even' :: Integral a => a -> Bool
even' 0 = True
even' n = odd' (n - 1)

odd' :: Integral a => a -> Bool
odd' 0 = False
odd' n = even' (n - 1)
```

Tail recursion: a special case

Definition (Tail recursion)

A function is *tail recursive* if the *last result of a recursive call* is the result of the function itself.

Loosely, the last thing a tail recursive function does - after having finished all other computations - is call itself with new arguments, or return a value.

- Such functions are useful because they have a trivial translation into loops
- Some languages (e.g. Scheme) *guarantee* that a tail recursive call will be transformed into a “loop-like” implementation using a technique called *tail call elimination*.

⇒ complexity remains unchanged, but implementation is more efficient.

- In Haskell implementations, while nice, this is not so important (other techniques are used)

Iteration ⇔ tail recursion

Loops are convenient

```
def factorial(n):
    res = 1
    for i in range(n, 1, -1):
        res *= i
    return res
```

Tail recursive implementation

- We can't write this directly, since we're not allowed to mutate things
- We can write it with a helper recursive function where all loop variables become arguments to the function

```
factorial n = loop n 1
  where loop n res | n < 0     = undefined
                   | n > 1     = loop (n - 1) (res * n)
                   | otherwise = res
```

Examples

Not tail recursive

Calls `(*)` after recursing

```
product' :: Num a => [a] -> a
product' [] = 1
product' (x:xs) = x * product' xs
```

Tail recursive

Recursive call to `loop` calls itself "outermost"

```
product' :: Num a => [a] -> a
product' xs = loop xs 1
  where loop [] n     = n
        loop (x:xs) n = loop xs (x * n)
```

Examples

Also for mutual recursion

Our `even/odd` functions are mutually tail recursive

```
even 0 = True
even n = odd (n-1)
odd 0 = False
odd n = even (n-1)

odd 4
==> even 3
==> odd 2
==> even 1
==> odd 0
==> False
```

Complexity

Is this a good implementation?

- The reverse of a list is computed by appending the head onto the reverse of the tail.

```haskell
reverse' :: [a] -> [a]
reverse' []     = []
reverse' (x:xs) = reverse' xs ++ [x]
```

Complexity

Is this a good implementation?

- The reverse of a list is computed by appending the head onto the reverse of the tail.

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

```
reverse' [1, 2, 3]
== reverse' [2, 3] ++ [1]                  -- applying reverse'
== (reverse' [3] ++ [2]) ++ [1]            -- applying reverse'
== ((reverse' [] ++ [3]) ++ [2]) ++ [1]    -- base case
== (([] ++ [3]) ++ [2]) ++ [1]             -- applying (++)
== ([3] ++ [2]) ++ [1]                     -- applying (++)
== [3, 2] ++ [1]                           -- applying (++)
== [3, 2, 1]
```

Complexity

Is this a good implementation?

- The reverse of a list is computed by appending the head onto the reverse of the tail.

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

```
reverse' [1, 2, 3]
== reverse' [2, 3] ++ [1]                   -- applying reverse'
== (reverse' [3] ++ [2]) ++ [1]             -- applying reverse'
== ((reverse' [] ++ [3]) ++ [2]) ++ [1]     -- base case
== (([] ++ [3]) ++ [2]) ++ [1]              -- applying (++)
== ([3] ++ [2]) ++ [1]                      -- applying (++)
== [3, 2] ++ [1]                            -- applying (++)
== [3, 2, 1]
```

- Recall that `(++)` must *traverse* its first argument
- So this implementation is $\mathcal{O}(n^2)$ in the length of the input list

Complexity

A more efficient way: combine reverse and append

```
-- helper function
reverse'' :: [a] -> [a] -> [a]
reverse'' [] ys     = ys
reverse'' (x:xs) ys = reverse'' xs (x:ys)

reverse' :: [a] -> [a]
reverse' xs = reverse'' xs []
```

Complexity

A more efficient way: combine reverse and append

```
-- helper function
reverse'' :: [a] -> [a] -> [a]
reverse'' [] ys     = ys
reverse'' (x:xs) ys = reverse'' xs (x:ys)

reverse' :: [a] -> [a]
reverse' xs = reverse'' xs []

reverse' [1, 2, 3, 4]
== reverse'' [1, 2, 3, 4] []  -- applying reverse'
== reverse'' [2, 3, 4] (1:[]) -- applying reverse''
== reverse'' [3, 4] (2:1:[])  -- applying reverse''
== reverse'' [4] (3:2:1:[])   -- applying reverse'
== reverse'' [] (4:3:2:1:[])  -- base case
== (4:3:2:1:[])               -- applying (:)
== [4, 3, 2, 1]
```

- Since `(:)` is $\mathcal{O}(1)$, this implementation is $\mathcal{O}(n)$.

Debugging errors

- Easy to get confused writing recursive functions
- The case enumeration is useful
- Helpful to write out the call stack “by hand” for a small example
- Usual error is that not all base cases are covered

Summary

- Contrasted sum and product types, and availability in other languages
- Discussed the pros and cons of classes and algebraic data types
- Considered how to write recursive functions
- Classified recursive functions