Functional Programming

2021-2022

Sjaak Smetsers

Loose ends

Lecture 7

Outline

- polymorphic type inference
- generic programming
- case study: Countdown Problem

Polymorhic type inference

- How do you solve the type inference puzzle puzzle?
- Before typing a function `f`, examine all functions used by `f` first.
  - start with a general type for each function
  - use patterns, guards and right-hand sides to derive more specific type information
  - introduce a fresh type for each polymorphic function (a new placeholder for each type variable)
- Type inference yields the most general type (MGT) of a function: Every valid type signature for a function is an instance of its MGT.

Polymorphic type inference: `twice`

```
twice :: …
twice f x = f (f x)
```

Polymorphic type inference: `twice`

```
twice :: ❶ ⟶ ❷ ⟶ ❸
twice f x = f (f x)
```

```
f   :: ❶ = ?
x   :: ❷ = ?
rhs :: ❸ = ?
```

Polymorphic type inference: `twice`

```
twice :: (4 ⟶ 5) ⟶ 4 ⟶ 3
twice f x = f @ (f @ x)
```

- Making the invisible application operator visible

```
@ :: (a⟶b) ⟶ a ⟶ b
```

```
f   :: 1 = ?
x   :: 2 = ?
rhs :: 3 = ?
@   :: (4 ⟶ 5)⟶ 4 ⟶ 5
1 = 4 ⟶ 5
2 = 4
```

Polymorphic type inference: `twice`

```
@ :: (a⟶b) ⟶ a ⟶ b
```

```
twice :: (6 ⟶ 7) ⟶ 6 ⟶ 7
twice f x = f @ (f @ x)
```

```
f   :: 1 = ?
x   :: 2 = ?
rhs :: 3 = 7
@   :: (6 ⟶ 7)⟶ 6 ⟶ 7
1 = 6 ⟶ 7
2 = 6
@   :: (6 ⟶ 7)⟶ 6 ⟶ 7
4 ⟶ 5 = 6 ⟶ 7
5 = 7
4 = 6
```

Polymorphic type inference: `twice`

```
twice :: (7 ⟶ 7) ⟶ 7 ⟶ 7
twice f x = f @ (f @ x)
```

```
@ :: (a⟶b) ⟶ a ⟶ b
```

```
f   :: 1 = ?
x   :: 2 = ?
rhs :: 3 = 7
@   :: (6 ⟶ 7)⟶ 6 ⟶ 7
1 = 6 ⟶ 7
2 = 6
@   :: (6 ⟶ 7)⟶ 6 ⟶ 7
4 ⟶ 5 = 6 ⟶ 7
5 = 7
4 = 6
6 = 7
```

Polymorphic type inference: `twice`

```
@ :: (a⟶b) ⟶ a ⟶ b
```

```
twice :: (7 ⟶ 7) ⟶ 7 ⟶ 7
twice f x = f @ (f @ x)
```

- No further restrictions: 7 remains to be 'unknown'

```
f   :: 1 = ?
x   :: 2 = ?
rhs :: 3 = 7
@   :: (4 ⟶ 6)⟶ 4 ⟶ 6
1 = 4 ⟶ 5
2 = 4
@   :: (6 ⟶ 7)⟶ 6 ⟶ 7
4 ⟶ 5 = 6 ⟶ 7
5 = 6
4 = 7
5 = 7
```

Polymorphic type inference: `twice`

```
twice :: (a ⟶ a) ⟶ a ⟶ a
twice f x = f (f x)
```

Polymorphic type inference: f6 (previous exam)

```
f6 :: ?
f6 xs = reverse [(y,x) | (x,y) ← xs]
```

Polymorphic type inference: f6 (previous exam)

```
f6 :: ❶ ⟶ ❷
f6 xs = reverse [(y,x) | (x,y) ⟵ xs]
```

```
reverse :: [a] ⟶ [b]
```

```
xs    :: ❶ = ?
rhs   :: ❷ = ?
left ⟵  :: ❸
right ⟵  :: [❸]
❶ = [❸]
```

Polymorphic type inference: f6 (previous exam)

```
f6 :: [❸] ⟶ ❷
f6 xs = reverse [(y,x) | (x,y) ⟵ xs]
```

```
reverse :: [a] ⟶ [b]
```

```
xs    :: ❶ = ?
rhs   :: ❷ = ?
left ⟵  :: ❸
right ⟵  :: [❸]
❶ = [❸]
❸ = (❹,❺)
x    :: ❹ = ?
y    :: ❺ = ?
```

Polymorphic type inference: f6 (previous exam)

```
f6 :: [(❹,❺)] ⟶ [❻]
f6 xs = reverse [(y,x) | (x,y) ⟵ xs]
```

```
reverse :: [a] ⟶ [a]
```

```
xs    :: ❶ = ?
rhs   :: ❷ = [❻]
left ⟵ :: ❸
right ⟵ :: [❸]
❶ = [❸]
❸ = (❹,❺)
x    :: ❹ = ?
y    :: ❺ = ?
reverse :: [❻]⟶[❻]
```

Polymorphic type inference: f6 (previous exam)

```
f6 :: [(④,⑤)] ⟶ [(⑤, ④)]
f6 xs = reverse [(y,x) | (x,y) ⟵ xs]
```

```
reverse :: [a] ⟶ [a]
```

```
xs     :: ① = ?
rhs    :: ② = [⑥]
left ⟵  :: ③
right ⟵  :: [③]
① = [③]
③ = (④,⑤)
x    :: ④ = ?
y    :: ⑤ = ?
reverse  :: [⑥]⟶[⑥]
⑥ = (⑤, ④)
```

Polymorphic type inference: f6 (previous exam)

```
fd :: [(a,b)] ⟶ [(b,a)]
f6 xs = reverse [(y,x) | (x,y) ⟵ xs]
```

Generic programming

- defining functions that can operate on a large range of datatypes
- eg defining instances of a class for different types at once
- idea: a new data type is expressed in terms of
  - unit: `() = ()`
  - product: `(a,b) = (a,b)`
  - sum: `Either a b = Left a | Right b`
- suppose we have a class C, and instances of for unit, product and sum
- by expressing a data type T in terms of unit, product and sum we get a T instance of C for free

Generic programming — representing data types

- We have base types `()`, `(,)`, `Either`. Let D be an (algebraic) data type. How do we express D in terms of these base types?
  - In case D is recursive we only specify 'one layer' of that type, ie. everything upto the recursive occurrences.
  - If D contains multiple constructors, use an `Either` for each additional constructor
  - for each constructor:
    - If it is constant (no arguments) use `()`.
    - If it has multiple arguments, use an `(,)` for each additional argument
- eg `data Colour = Red | Green | Blue`
- can be described by the following type: `type COLOUR = Either (Either ()()) ()`
- conversions:

```
toCOLOUR :: Colour ⟶ COLOUR
toCOLOUR Red   = Left (Left  ()) -- 00
toCOLOUR Green = Left (Right ()) -- 01
toCOLOUR Blue  = Right ()        -- 1

fromCOLOUR :: COLOUR ⟶ Colour
fromCOLOUR (Left (Left  ())) = Red
fromCOLOUR (Left (Right ())) = Green
fromCOLOUR (Right ())        = Blue
```

GP: representing data types (II)

- More examples

```
data Card = Faceless Int | Jack | Queen | King
type CARD = Either (Either Int ()) (Either () ())

data BTree a = Leaf a | Bin (BTree a) (BTree a)
type BTREE a = Either a (BTree a, BTree a)
```

- conversions:

```
toCARD :: Card ⟶ CARD
toCARD (Faceless n) = Left  (Left n)
toCARD Jack         = Left  (Right ())
toCARD Queen        = Right (Left ())
toCARD King         = Right (Right ())
```

```
fromCARD :: CARD ⟶ Card
fromCARD (Left  (Left n))    = Faceless n
fromCARD (Left  (Right ())) = Jack
fromCARD (Right (Left ()))  = Queen
fromCARD (Right (Right ())) = King
```

```
toBTREE :: BTree a ⟶ BTREE a
toBTREE (Leaf e)  = Left e
toBTREE (Bin l r) = Right (l,r)
```

```
fromBTREE :: BTREE a ⟶ BTree a
fromBTREE (Left e)       = Leaf e
fromBTREE (Right (l,r)) = Bin l r
```

GP: free instances

- suppose we have

```
instance Eq () where
   () == () = True
instance (Eq a, Eq b) ⇒ Eq (a, b) where
   (x1,y1) == (x2,y2) = x1 == x2 && y1 == y2
instance (Eq a, Eq b) ⇒ Eq (Either a b) where
   Left  l1 == Left l2  = l1 == l2
   Right r1 == Right r2 = r1 == r2
   _        == _        = False
```

- then

```
instance Eq Colour where
   c1 == c2 = toCOLOUR c1 == toCOLOUR c2
instance Eq Card where
   c1 == c2 = toCARD c1 == toCARD c2
instance (Eq a) ⇒ Eq (BTree a) where
   t1 == t2 = toBTREE t1 == toBTREE t2
```

GP: Rose trees

- multi-way trees aka rose trees

```
data RTree a = Branch a [RTree a]
type RTREE a = (a, [RTree a])

toRTREE :: RTree a ⟶ RTREE a
toRTREE (Branch e trs) = (e, trs)

fromRTREE :: RTREE a ⟶ RTree a
fromRTREE (e, trs)  = (Branch e trs)
```

- equality

```
instance (Eq a) ⟹ Eq (RTree a) where
   t1 == t2 = toRTREE t1 == toRTREE t2
```

GP: Serialization

- Converting data structures into a bit string

```
type Bit = Int
class Serialize a where
  compress   :: a ⟶ [Bit]
  decompress :: [Bit] ⟶ (a,[Bit])
```

- `decompress` might not consume the whole input; the remainder of input is returned as well.
- auxiliary functions

```
compressNum :: Int ⟶ Int ⟶ [Bit]
compressNum 0 _n = []
compressNum s n
  | n `mod` 2 == 0 = 0 : compressNum (s-1)  (n `div` 2)
  | otherwise      = 1 : compressNum (s-1)  (n `div` 2)

decompressNum :: Int ⟶ [Bit] ⟶ (Int,[Bit])
decompressNum 0 bs     = (0, bs)
decompressNum n (b:bs) = let (dn, rbs) = decompressNum (n-1) bs in (b + 2*dn, rbs)
```

GP: Instances

```
instance Serialize Bool where
  compress False = [0]
  compress True  = [1]

  decompress (b:bs)
     | b == 0    = (False, bs)
     | otherwise = (True, bs)
instance Serialize Char where
  compress c    = compressNum 8 (ord c)
  decompress bs = let (dn, rbs) = decompressNum 8 bs in (chr dn, rbs)
instance Serialize Int where
  compress i    = compressNum 32 i
  decompress bs = decompressNum 32 bs
```

GP: Base instances

```
instance Serialize () where
   compress ()    = []
   decompress bs = ((), bs)
instance (Serialize a, Serialize b) ⇒ Serialize (a,b) where
   compress (x,y)    = compress x ++ compress y
   decompress bs     = let (x, xbs) = decompress bs
                           (y, ybs) = decompress xbs
                       in ((x,y), ybs)
instance (Serialize a, Serialize b) ⇒ Serialize (Either a b) where
   compress (Left l)     = 0 : compress l
   compress (Right r)    = 1 : compress r

   decompress (0:bs)     = let (l, lbs) = decompress bs in (Left l, lbs)
   decompress (1:bs)     = let (r, rbs) = decompress bs in (Right r, rbs)
```

GP: user-defined instances

- All other instances we get for free

```
instance Serialize Card where
    compress = compress . toCARD
    decompress bs = let (d, dbs) = decompress bs in (fromCARD d, dbs)

instance (Serialize a) => Serialize [a] where
    compress = compress . toLIST
    decompress bs = let (d, dbs) = decompress bs in (fromLIST d, dbs)

instance (Serialize a) => Serialize (RTree a) where
    compress = compress . toRTREE
    decompress bs = let (d, dbs) = decompress bs in (fromRTREE d, dbs)
```

Case study: Countdown

- A popular quiz programme on British television that has been running for almost 20 years.

- Based upon an original French version called "Des Chiffres et Des Lettres".

- Includes a numbers game that we shall refer to as the countdown problem.

Example

- Using the numbers

1 3 7 10 25 50

- and the arithmetic operators

+ - * ÷

- construct an expression whose value is 765

Rules

- All the numbers, including intermediate results, must be integers greater than zero.
- Each of the source numbers can be used at most once when constructing the expression.
- For every source number n: $1 \leq n \leq 100$
- The target number is greater than 100

Solution

- For our example, one possible solution is

$\boxed{(25-10) * (50+1)}$ = $\boxed{765}$

- Notes:
  - There are <u>780</u> solutions for this example.
  - Changing the target number to $\boxed{831}$ gives an example that has <u>no</u> solutions.

Operators

- Operators:

```
data Op = Add | Sub | Mul | Div
```

- Apply an operator:

```
apply :: Op → Int → Int → Int
apply Add x y = x + y
apply Sub x y = x - y
apply Mul x y = x * y
apply Div x y = x `div` y
```

- Determine whether the result of applying an operator to two integers greater than zero satisfies the rules:

```
valid :: Op → Int → Int → Bool
valid Add _ _ = True
valid Sub x y = x > y
valid Mul _ _ = True
valid Div x y = x `mod` y == 0
```

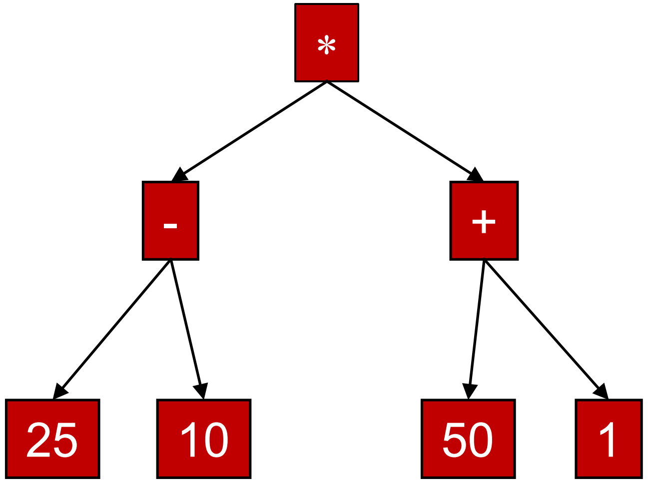
Expressions

- Expressions

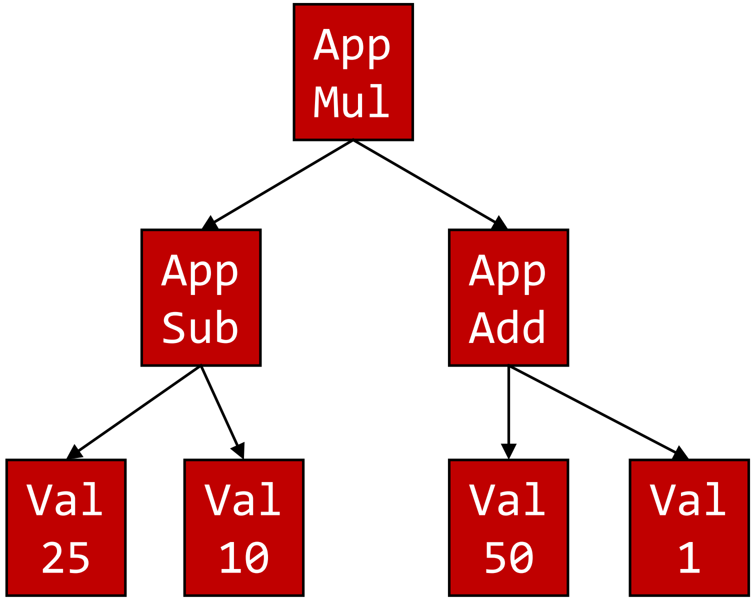
```
data Expr = Val Int | App Op Expr Expr
```

- Expressions are trees

(25-10) * (50+1)




```
App Mul (App Sub (Val 25) (Val 10))
        (App Add (Val 50) (Val 1 ))
```



Evaluation

- Return the overall value of an expression

```
eval :: Expr → [Int]
eval (Val n)     = [n]
eval (App o l r) = [apply o x y | x ← eval l
                                , y ← eval r
                                , valid o x y]
```

Either succeeds and returns a singleton list, or fails and returns the empty list.

Formalising The Problem

- Return a list of all possible ways of choosing two or more elements from a list:

```
choices :: [a] → [[a]]
```

- For example:

```
> choices [1..3]
[[1,2],[2,1],[1,3],[3,1],[2,3],[3,2],[1,2,3],[2,1,3],
[3,2,1],[2,3,1],[3,1,2],[1,3,2]]
```

Formalising The Problem (2)

- Return a list of all the values in an expression:

```
values :: Expr → [Int]
values (Val n)     = [n]
values (App _ l r) = values l ++ values r
```

- Decide if an expression is a solution for a given list of source numbers and a target number:

```
solution :: Expr → [Int] → Int → Bool
solution e ns n = values e `elem` choices ns
                  && eval e == [n]
```

Brute Force Implementation

- build a list of all possible expressions whose values are precisely a given list of numbers:

```
exprs :: [Int] → [Expr]
exprs [n] = [Val n]
exprs ns  = [App o l r | m ← [1..length ns-1],
                         let (ls,rs) = splitAt m ns,
                         l ← exprs ls,
                         r ← exprs rs,
                         o ← [Add,Sub,Mul,Div]]
```

The key function in this example.

Solving the problem

- Return a list of all possible expressions that solve an instance of the countdown problem:

```
solutions :: [Int] → Int → [Expr]
solutions ns n = [e | ns' ← choices ns,
                      e   ← exprs ns’,
                      eval e == [n]]
```

How Fast Is It?

System:	Intel(R) Core(TM) i7-10510U CPU 4.3 GHz
Compiler:	GHC version 8.10.3
Example:	`solutions [1,3,7,10,25,50] 765`
One solution:	0.16 seconds
All solutions:	5.44 seconds

Can We Do Better?

- Many of the expressions that are considered will typically be invalid - fail to evaluate.
  - For our example, only around 5 million of the 33 million possible expressions are valid.
- Combining generation with evaluation would allow earlier rejection of invalid expressions.
- Many expressions will be *essentially the same* using simple arithmetic properties, such as:

```
x * y = y * x
x * 1 = x = 1 * x
```

Exploiting Properties

- Strengthening the `valid` predicate to take account of commutativity and identity properties:

```
valid :: Op → Int → Int → Bool
valid Add x y  = x ≤ y
valid Sub x y  = x > y
valid Mul x y  = x ≤ y && x ≠ 1 && y ≠ 1
valid Div x y  = x `mod` y == 0 && y ≠ 1
```

Imporoving the implementation

- We seek to define a function that fuses together the generation and evaluation of expressions:

```
exprsF :: [Int] → [(Expr,Int)]
exprsF [n] = [(Val n, n)]
exprsF ns  = [(App o l r, apply o vl vr) |
                          m             ← [1..length ns-1],
                          let (ls,rs) = splitAt m ns,
                          (l,vl)        ← exprsF ls,
                          (r,vr)        ← exprsF rs,
                          o             ← [Add,Sub,Mul,Div],
                          valid o vl vr
                          ]
```

How Fast Is It Now?

- example: `solutions' [1,3,7,10,25,50] 765`
- solutions: 49 expressions

Around 16 times less.

- finding all solutions: 0.084 seconds

Around 65 times faster.