计算概论A—实验班

函数式程序设计

Functional Programming

胡振江，张 伟

北京大学 计算机学院

2023年09～12月

Adapted from Graham’s Lecture slides

第9章：An Example: The Countdown Problem

What Is Countdown?

- A popular quiz programme on British television that has been running since 1982.
- 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.

Countdown: An example

- Using the numbers

1	3	7	10	25	50

and the arithmetic operators

+	−	*	÷

construct an expression whose value is **765**

Countdown: Two Rules

1. All the numbers, including intermediate results, must be positive naturals (1,2,3,…).
2. Each of the source numbers can be used at most once when constructing the expression.

Countdown: The example

- For the example above, one possible solution is

```
(25 - 10) * ( 50 + 1)  =  765
```

- There are 780 solutions for this example.
- Changing the target number to **831** gives an example that has no solutions.

Evaluating Expressions

- A type for Operators:

```
data Op = Add | Sub | Mul | Div deriving (Show)
```

- 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
```

- Decide if the result of applying an operator to two positive natural numbers is another such:

```
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
```

- A type for Expressions:

```
data Expr = Val Int | App Op Expr Expr
          deriving (Show)
```

✤ Return the overall value of an expression, provided that it is a positive natural number:

```
eval :: Expr -> [Int]
eval (Val n)     = [n | n > 0]
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

Some combinatorial functions

- Returns all subsequences of a list.

```
subs :: [a] -> [[a]]
subs [] = [[]]
subs (x:xs) = let yss = subs xs
              in yss ++ map (x:) yss
```

```
> subs [1,2,3]
[[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]
```

Some combinatorial functions

- Returns all possible ways of inserting a new element into a list.

```
interleave :: a -> [a] -> [[a]]
interleave x [] = [[x]]
interleave x (y:ys) = (x:y:ys) : map (y:) (interleave x ys)
```

```
> interleave 1 [2,3,4]
[[1,2,3,4],[2,1,3,4],[2,3,1,4],[2,3,4,1]]
```

Some combinatorial functions

- Returns all permutations of a list.

```
perms :: [a] -> [[a]]
perms []      = [[]]
perms (x:xs) = concat $ map (interleave x) (perms xs)
```

```
> perms [1,2,3]
[[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]
```

Some combinatorial functions

- Return a list of all possible ways of choosing zero or more elements from a list in any order.

```haskell
choices :: [a] -> [[a]]
choices = concat . map perms . subs
```

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

Formalising The Problem

- 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 Solution

- Return a list of all possible ways of splitting a list into two non-empty parts.

```
split :: [a] -> [([a],[a])]
split []  = []
split [_] = []
split (x:xs) = ([x],xs) : [ (x:ls, rs) | (ls,rs) <- split xs ]
```

```
> split [1,2,3,4]
[([1],[2,3,4]),([1,2],[3,4]),([1,2,3],[4])]
```

Brute Force Solution

- Return a list of all possible expressions whose values are precisely a given list of numbers.

```
exprs :: [Int] -> [Expr]
exprs []  = []
exprs [n] = [Val n]
exprs ns  = [e | (ls,rs) <- split ns
               ,       l <- exprs ls
               ,       r <- exprs rs
               ,       e <- combine l r]
```

```
combine :: Expr -> Expr -> [Expr]
combine l r = [App o l r | o <- [Add,Sub,Mul,Div]]
```

Brute Force Solution

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

```haskell
solutions :: [Int] -> Int -> [Expr]
solutions ns n = [e | ns' <- choices ns
                    ,  e  <- exprs ns'
                    , eval e == [n]]
```

How Fast Is It?

Hardware:	2.8GHz Core 2 Duo, 4GB RAM
Compiler:	GHC version 7.10.2
Example:	solutions [1,3,7,10,25,50] 765
One solution:	0.108 seconds
All solutions:	12.224 seconds

如果在ghci中运行，时间估计会增加一个数量级

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.

Fusing generation and evaluation

- A type for Valid expressions and their values:

```
type Result = (Expr, Int)
```

- A function without fusion

```
results :: [Int] -> [Result]
results ns = [(e,n) | e <- exprs ns
                    , n <- eval  e]
```

Fusing generation and evaluation

- ✤ A function without fusion

```
results :: [Int] -> [Result]
results ns = [(e,n) | e <- exprs ns
                    , n <- eval  e]
```

- ✤ A function with fusion

```
results :: [Int] -> [Result]
results []  = []
results [n] = [(Val n, n) | n > 0]
results ns  = [res | (ls,rs) <- split ns
                   ,      lx  <- results ls
                   ,      ry  <- results rs
                   ,     res  <- combine' lx ry]

combine' :: Result -> Result -> [Result]
combine' (l,x) (r,y) = [(App o l r, apply o x y)
                       | o <- [Add,Sub,Mul,Div]
                       , valid o x y]
```

A better solution

```haskell
solutions' :: [Int] -> Int -> [Expr]
solutions' ns n = [e | ns'   <- choices ns
                     , (e,m) <- results ns'
                     , m == n]
```

How Fast Now?

Hardware: 2.8GHz Core 2 Duo, 4GB RAM			
Compiler: GHC version 7.10.2			
Example: solutions [1,3,7,10,25,50] 765			
One solution:	0.108 s	0.014 s	
All solutions:	12.224 s	1.312 s	
	Brute Force	Fusion	

Can We Do Better Further?

- Many expressions will be essentially the same using simple arithmetic properties, such as:

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

- Exploiting such properties would considerably reduce the search and solution spaces.

A better valid function

- In Haskell, a new name for an existing type can be defined using a type declaration.

```
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
```

How Fast Now?

Hardware: 2.8GHz Core 2 Duo, 4GB RAM			
Compiler: GHC version 7.10.2			
Example: solutions [1,3,7,10,25,50] 765			
One solution:	0.108 s	0.014 s	0.007 s
All solutions:	12.224 s	1.312 s	0.119 s
	Brute Force	Fusion	better valid

作业

作业

9-1

Modify the final program to:

1. allow the use of exponentiation in expressions;
2. produce the nearest solutions if no exact solution is possible;
3. order the solutions using a suitable measure of simplicity.

第9章：An Example: The Countdown Problem

就到这里吧