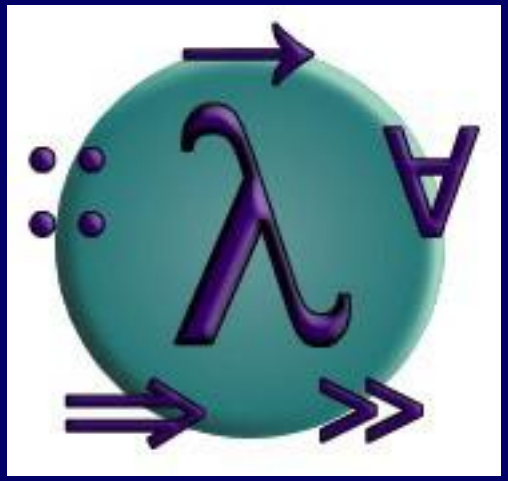
PROGRAMMING IN HASKELL
プログラミングHaskell



Chapter 11 - The Countdown Problem
切符番号遊び

愛知県立大学 情報科学部 計算機言語論(山本晋一郎・大久保弘崇、2013年)講義資料
オリジナルは http://www.cs.nott.ac.uk/~gmh/book.html を参照のこと

切符番号遊び（Countdown）とは

- 英国のテレビで 1982 年から放送されている人気のクイズ番組
- フランス版の "Des Chiffres et Des Lettres" がオリジナル
- そのクイズ番組に切符番号遊びと呼ばれる数を用いたゲームがあった

Example

以下の数

1　3　7　10　25　50

と算術演算子

＋　－　＊　÷

括弧も使って良い

を使って値が 765 になる式を作れるか?

ゲームのルール

- 計算の途中結果を含む全ての数は、正の自然数 (1, 2, 3, ...) であること
- 与えられた自然数を 2 回以上使ってはいけない
- テレビ番組では他のルールもあったが省略する

先ほどの例では、以下の式は正解の一つ

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

注意:

- 780 個の正解が存在する
- 値を `831` に変えると、正解は無い

式を評価する

演算子を表すデータ型 Op:

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

演算子を引数に適用する apply:

```
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
```

演算子を 2 つの自然数に適用した結果が、
自然数か否かを判定する述語 valid:

```
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
```

式を表すデータ型 Expr:

式は、自然数そのものか、
2 つの式への演算子の適用

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

式の値を求める eval
ただし、値は自然数であること:

```
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]
```

長さ 1 の自然数リストは成功、
空リストは失敗

進行

- 以下の順序で説明を行う

1. 問題の定式化
   - 正解か否かを判定する述語を定義する
2. わかりやすいが遅いプログラム
   - 全ての式 (解の候補) を順に生成し、正解か否か判定する
3. 効率化の手法
   - 無駄な解候補の生成を減らす

問題の形式化

リストから 0 個以上の要素を取り出す方法の順列を
リストにする choices:

```
choices :: [a] → [[a]]
```

For example:

```
> choices [1,2]

[[],[1],[2],[1,2],[2,1]]
```

式の中の全ての自然数をリストにする values:

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

式 e が、自然数リスト ns と目標の自然数 n で与えられる切符番号遊びの正解か判定する述語 solution:

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

実行例

総当たりな解法

リストを 2 つの非空リストに分割する方法のすべてをリストにする関数 split:

```
split :: [a] → [([a],[a])]
```

For example:

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

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

与えられた自然数の全部を使ってできる、全ての式をリストにする exprs:

```
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]
```

この章のキーとなる関数

2 つの式を、全ての演算子で繋ぐ combine:

```
combine     :: Expr → Expr → [Expr]
combine l r =
   [App o l r | o ← [Add,Sub,Mul,Div]]
```

切符番号遊びの全ての正解のリストを返す solutions:

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

どのくらいの速さか?

System: 1.2GHz Pentium M laptop

Compiler: GHC version 6.4.1

Example:

```
solutions [1,3,7,10,25,50] 765
```

One solution: 0.36 seconds

All solutions: 43.98 seconds

さらに良い方法は?

- 生成した多くの式の値が自然数ではないため無駄になる
- 例では、3300 万の式のうち 500 万だけが自然数
- 生成と評価を組み合わせれば、無駄な式を早期に除外できる

2 つの関数を融合する

正しい式とその値を組を表すデータ型 Result:

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

式の生成と評価を融合した関数 results:

```
results    :: [Int] → [Result]
results ns = [(e,n) | e ← exprs ns
                    , n ← eval e]
```

This behaviour is achieved by defining

```
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]
```

where

```
combine' :: Result → Result → [Result]
```

結果を結合する combine’:

```
combine' (l,x) (r,y) =
   [(App o l r, apply o x y)
      | o ← [Add,Sub,Mul,Div]
      , valid o x y]
```

切符番号遊びを解く新しい関数 solutions’:

```
solutions'      :: [Int] → Int → [Expr]
solutions' ns n =
   [e | ns'   ← choices ns
      , (e,m) ← results ns'
      , m == n]
```

速くなったか?

Example:

```
solutions' [1,3,7,10,25,50] 765
```

One solution: 0.04 seconds

All solutions: 3.47 seconds

Around 10 times faster in both cases.

さらなる高速化は?

- 多くの式は、算術式の性質により本質的に同じとみなせる、例えば:

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

```
x * 1  =  x
```

- この性質を用いると、探索すべき式と正解の数を減らすことができる

代数則の利用

- Add x y と Add y x はどちらか一方で十分
- Mul x y と Mul y x も同様
- さらに、Mul x 1 と Mul 1 y は、どちらも不要
- Div x 1 も同様

交換律と単位元を考慮して、正しい式をより制限する valid:

```
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
```

ふたたび、速くなったか?

Example: `solutions'' [1,3,7,10,25,50] 765`

Valid: 250,000 expressions

Around 20 times less.

Solutions: 49 expressions

Around 16 times less.

One solution: 0.02 seconds

Around 2 times faster.

All solutions: 0.44 seconds

Around 7 times faster.

テレビ番組で出される問題の 1 つの正解なら、瞬く間に求めることができる。全ての正解でも、1 秒以下だろう。