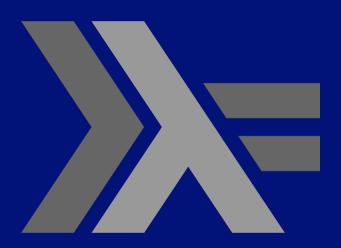
# PROGRAMMING IN HASKELL



Chapter 9 - The Countdown Problem

### What Is Countdown?

- A popular <u>quiz programme</u> on British television that has been running since 1982.
- Based upon an original <u>French</u> version called "Des Chiffres et Des Lettres".
- Includes a numbers game that we shall refer to as the <u>countdown problem</u>.

# **Example**

Using the numbers

1

3

7

10

25

50

and the arithmetic operators









construct an expression whose value is

## Rules

- All the numbers, including intermediate results, must be positive naturals (1,2,3,...).
- Each of the source numbers can be used at most once when constructing the expression.
- We <u>abstract</u> from other rules that are adopted on television for pragmatic reasons.

For our example, one possible solution is

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

#### Notes:

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

# **Evaluating Expressions**

#### 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
```

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
```

#### **Expressions:**

data Expr = Val Int | App Op Expr Expr

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

```
eval :: Expr \rightarrow [Int]
eval (Val n) = [n | n > 0]
eval (App o l r) = [apply o x y | x \leftarrow eval l
, y \leftarrow 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 zero or more elements from a list:

```
choices :: [a] \rightarrow [[a]]
```

## For example:

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

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 \rightarrow [Int] \rightarrow Int \rightarrow Bool solution e ns n = elem (values e) (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] \rightarrow [([a],[a])]
```

For example:

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

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

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

The key function in this lecture.

# Combine two expressions using each operator:

```
combine :: Expr → Expr → [Expr]
combine 1 r =
  [App o 1 r | o ← [Add, Sub, Mul, Div]]
```

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: 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

### Can We Do Better?

- Many of the expressions that are considered will typically be <u>invalid</u> - fail to evaluate.
- For our example, only around <u>5 million</u> of the 33 million possible expressions are valid.
- Combining generation with evaluation would allow <u>earlier rejection</u> of invalid expressions.

# **Fusing Two Functions**

Valid expressions and their values:

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

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

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

## This behaviour is achieved by defining

#### where

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

## Combining results:

New function that solves countdown problems:

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

# **How Fast Is It Now?**

Example:

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

One solution: 0.014 seconds

Julius

both cases.

Around 10

times faster in

All solutions: 1.312 seconds

# Can We Do Better?

Many expressions will be <u>essentially the same</u> using simple arithmetic properties, such as:

$$x * y = y * x$$
 $x * 1 = x$ 

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

# **Exploiting Properties**

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

```
valid :: Op \rightarrow Int \rightarrow Int \rightarrow Bool
valid Add x y = X \leq Y
valid Sub x y = x > y
valid Mul x y = X \leq Y \otimes X \times Y = 1 \otimes Y \times Y \times Y = 1 \otimes Y \times Y \times Y = 1 \otimes Y = 1 \otimes Y \times Y
```

# **How Fast Is It Now?**

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: 0.007 seconds

Around 2 times faster.

All solutions: 0.119 seconds

Around 11 times faster.

More generally, our program usually returns all solutions in a fraction of a second, and is around 100 times faster that the original version.