Functional Pearls

Christophe Scholliers

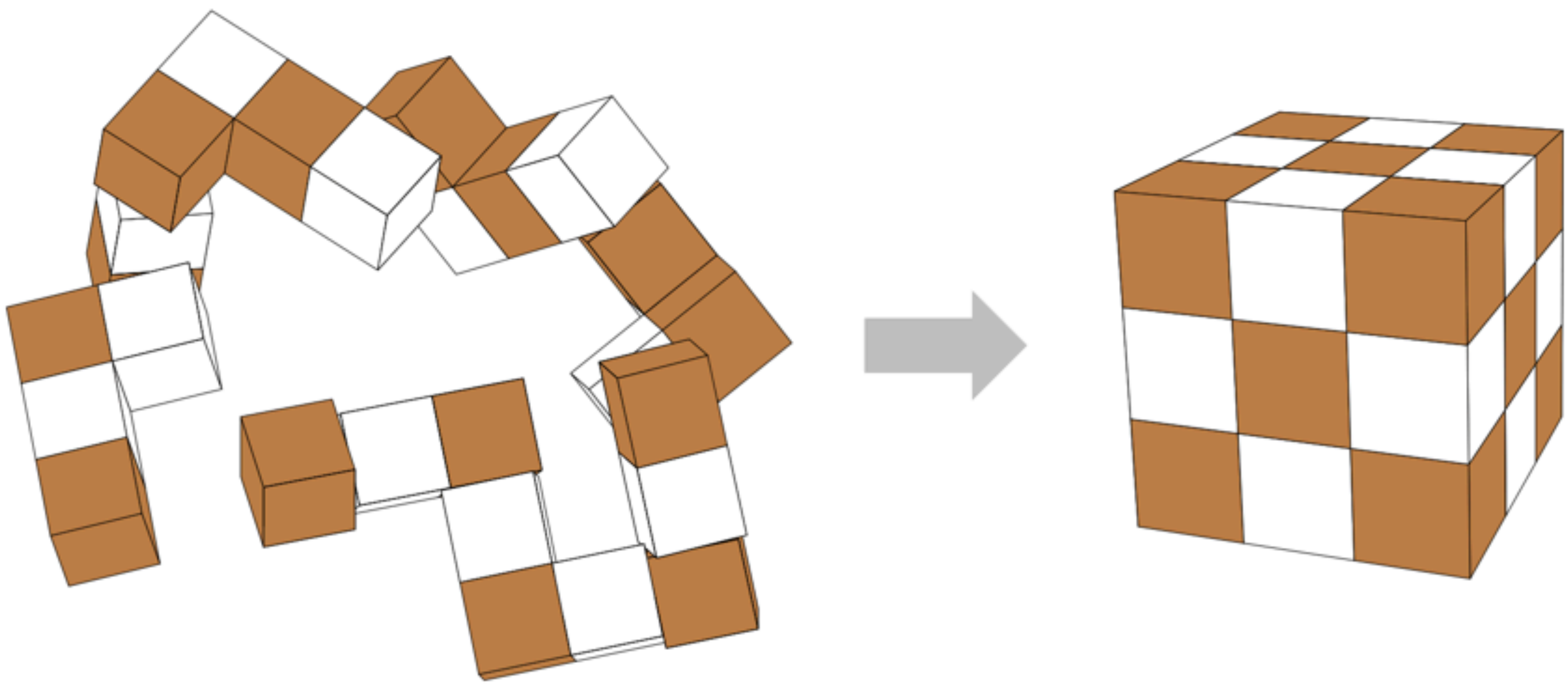
What are Functional Pearls ?

Functional pearls are elegant, instructive examples of functional programming. They are supposed to be fun, and they teach important programming techniques and fundamental design principles. They traditionally appear in The Journal of Functional Programming, and at ICFP and affiliated workshops.

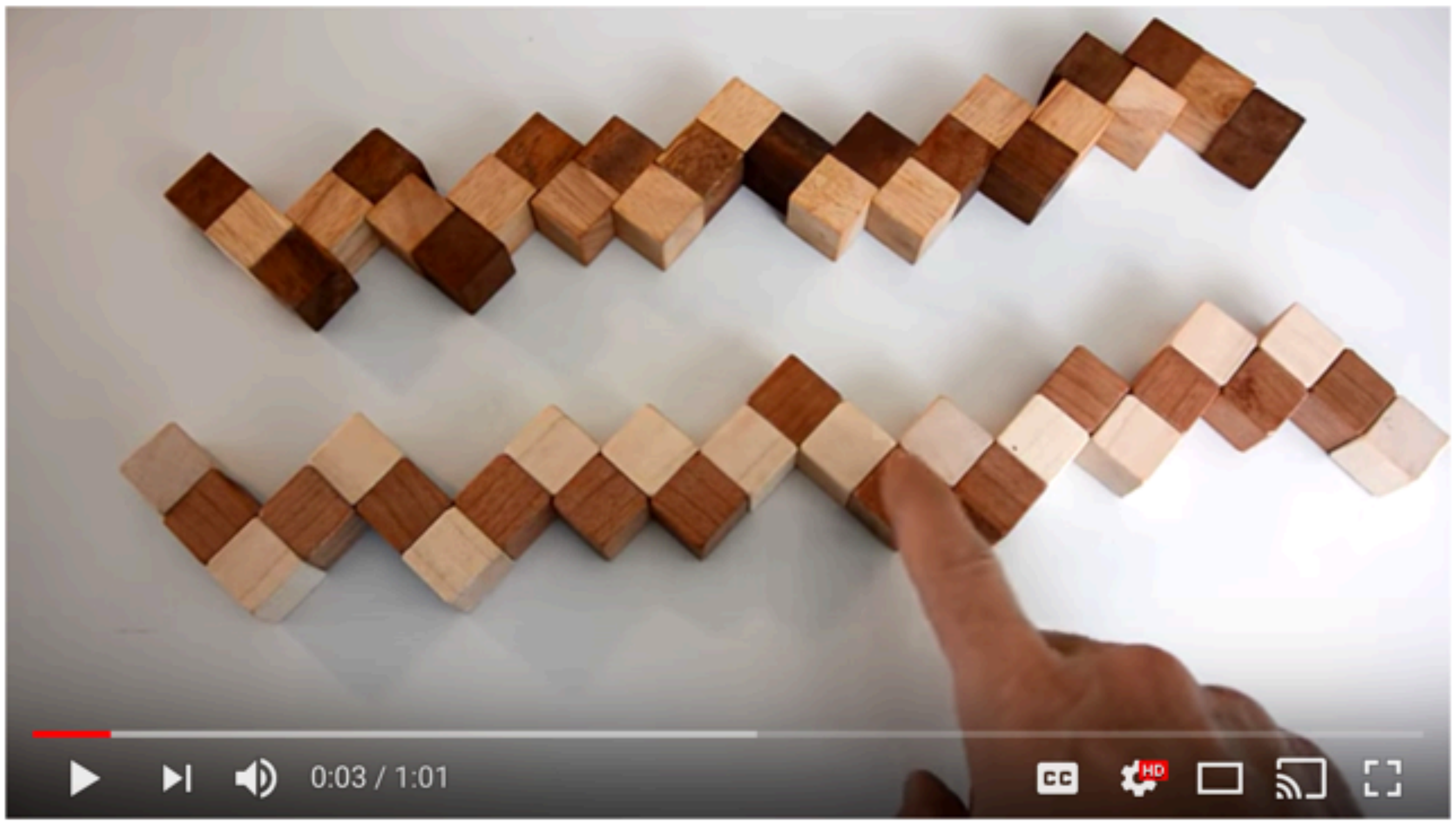
Solving the Snake Cube Puzzle in Haskell

MARK P. JONES Department of Computer Science
Portland State University, Portland, Oregon, USA

Snake Cube

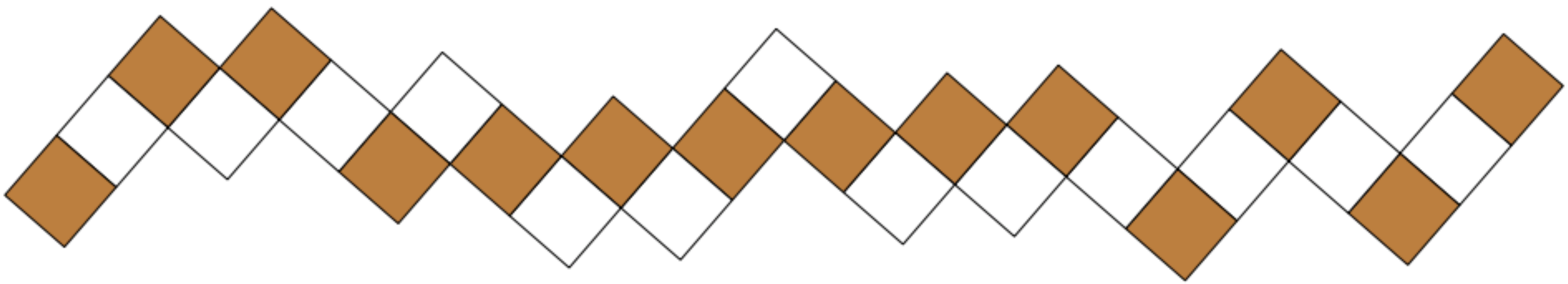


How to solve one



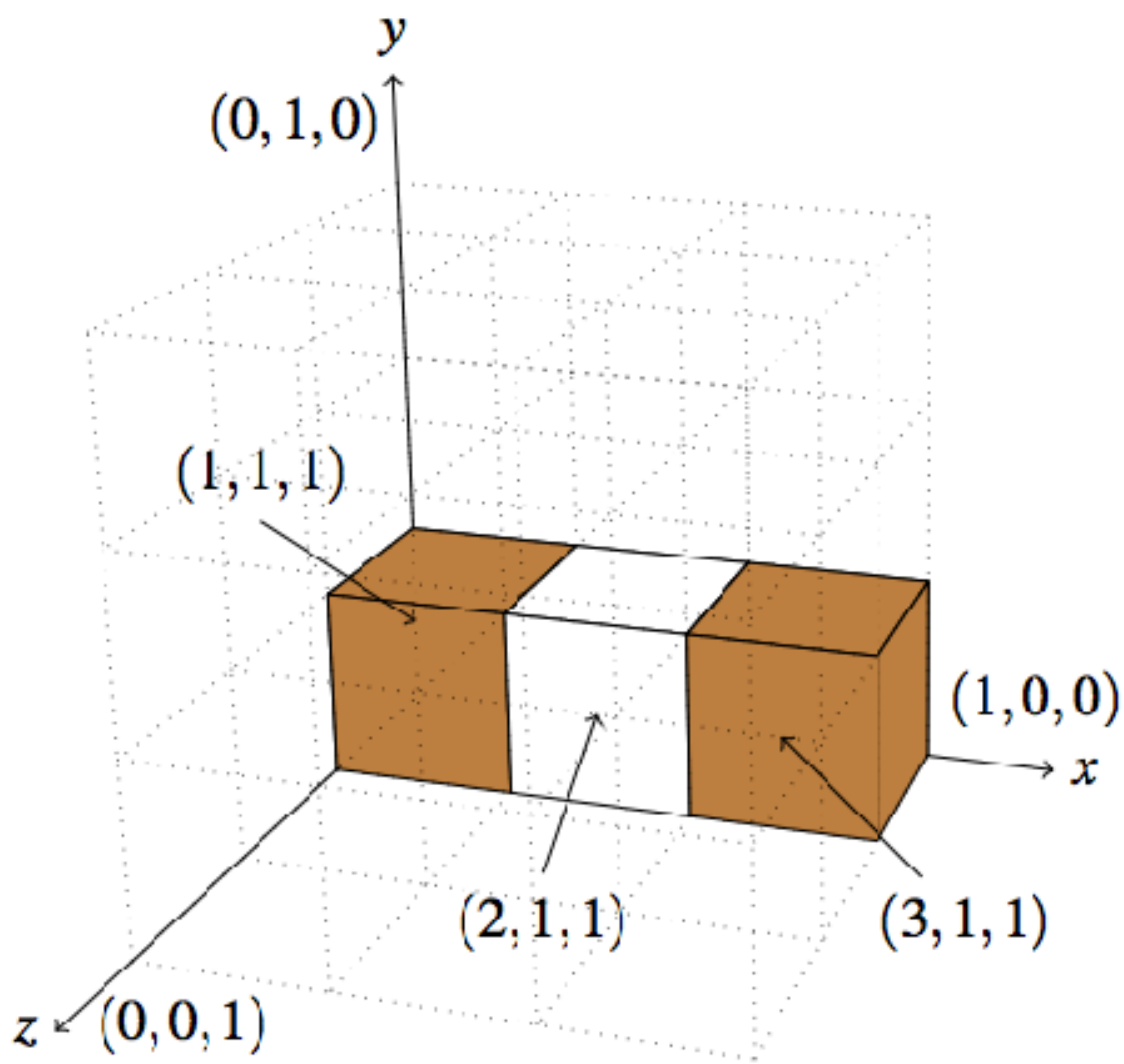
https://www.youtube.com/watch?v=iTzVPgFjE9c

Construction



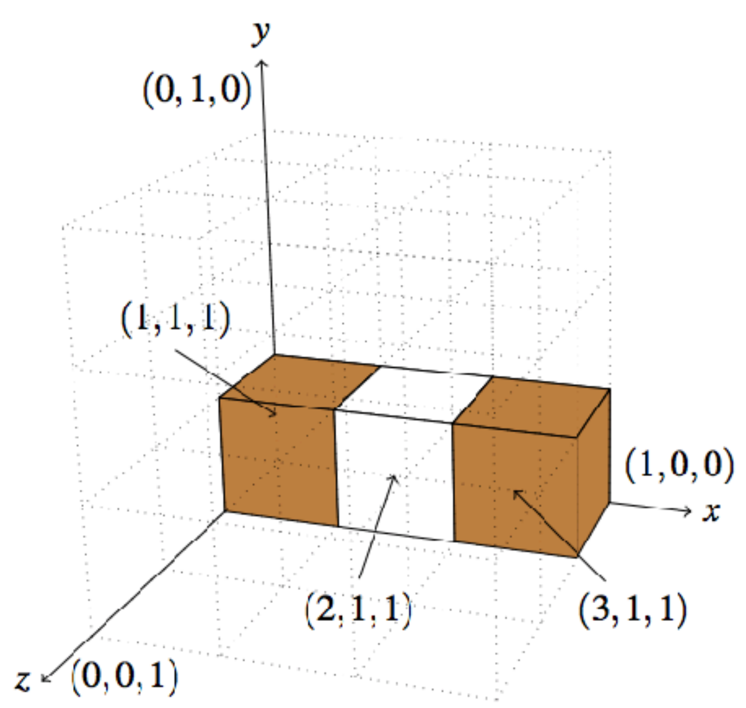
```
snake :: [Int]
snake  = [ 3, 2, 2, 3, 2, 3, 2, 2, 3, 3, 2, 2, 2, 3, 3, 3, 3 ]
```

Moving in to Three Dimensions

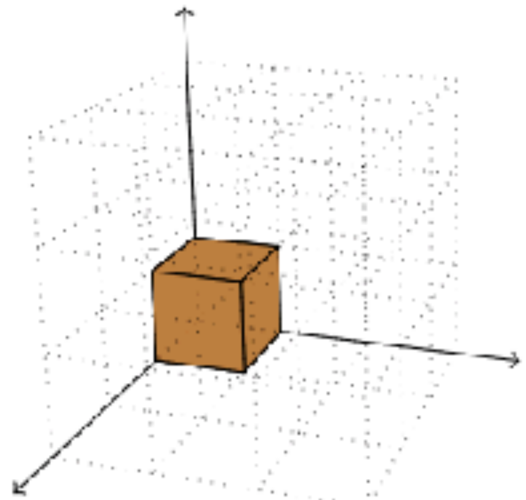


Moving in to Three Dimensions

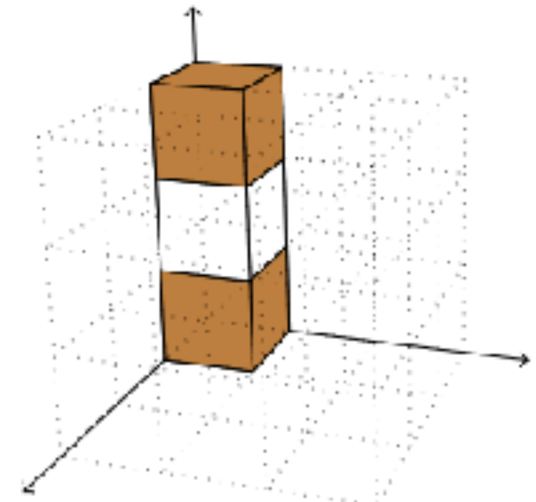
```
type Position  = (Int, Int, Int)
type Direction = (Int, Int, Int)

inCube                   :: Int -> Position -> Bool
inCube n (x, y, z) = valid x && valid y && valid z
                     where valid i = 1<=i && i<=n
```



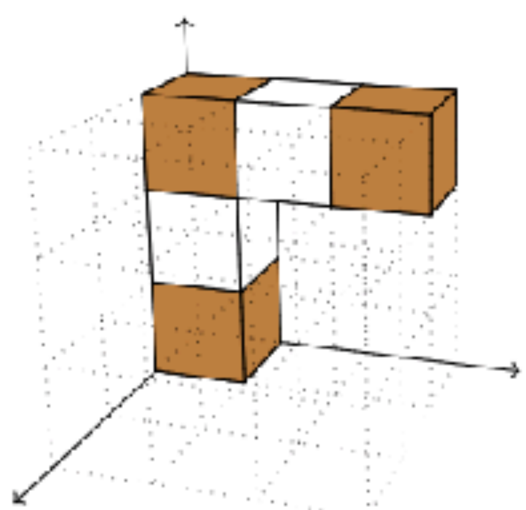
Describing Solutions



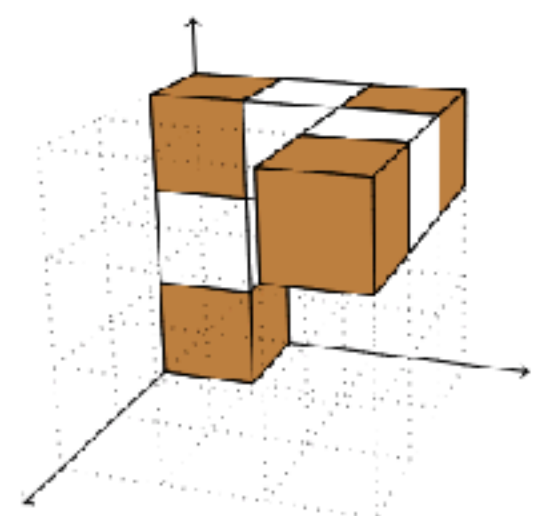
$soln_0 = [[(1, 1, 1)]]$



$soln_1 = [(1, 3, 1), (1, 2, 1)] : soln_0$



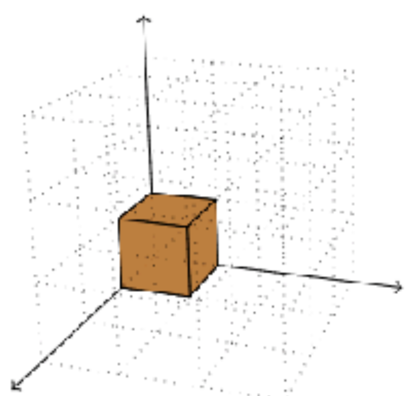
$soln_2 = [(3, 3, 1), (2, 3, 1)] : soln_1$



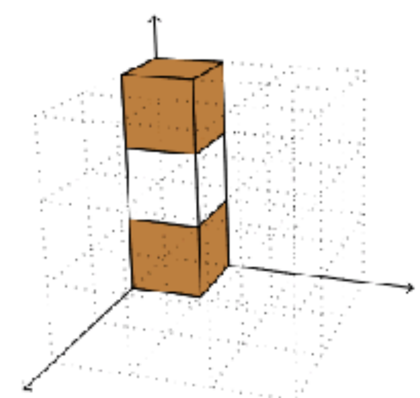
$soln_3 = [(3, 3, 3), (3, 3, 2)] : soln_2$

Describing Solutions

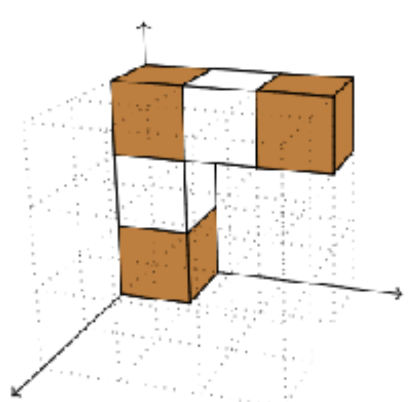
```
type Solution = [Section]
type Section  = [Position]
```



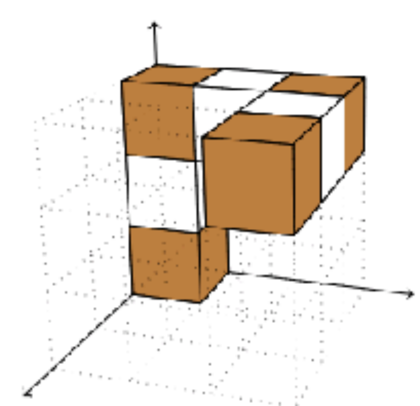
$soln_0 = [[(1,1,1)]]$



$soln_1 = [(1,3,1),(1,2,1)] : soln_0$



$soln_2 = [(3,3,1),(2,3,1)] : soln_1$



$soln_3 = [(3,3,3),(3,3,2)] : soln_2$

Building Sections

```
iterate :: (a -> a) -> a -> [a]
```

Source

#

`iterate f x` returns an infinite list of repeated applications of `f` to `x`:

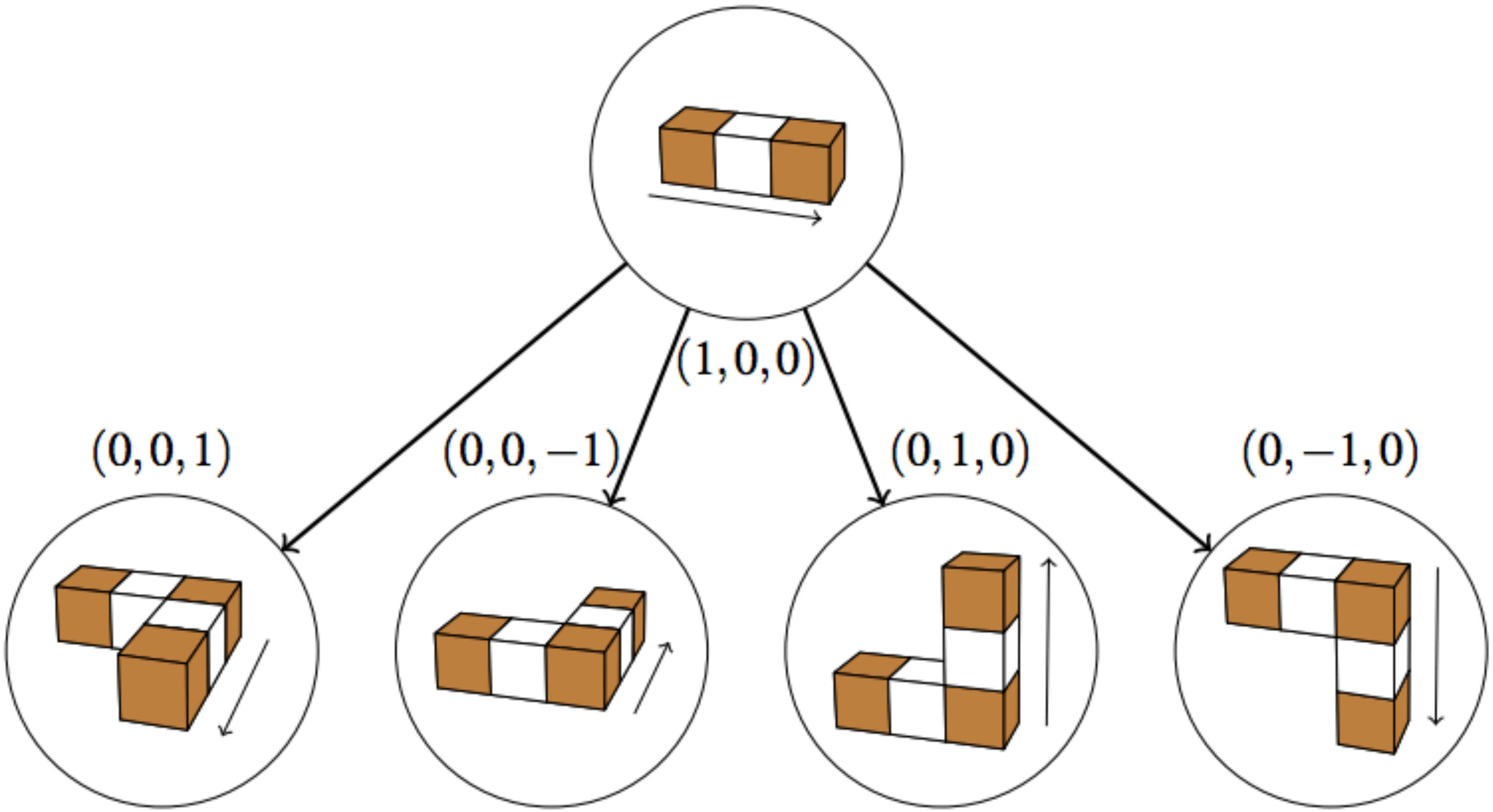
```
iterate f x == [x, f x, f (f x), ...]
```

```haskell
section                     :: Position -> Direction -> Int -> Section
section start (u,v,w) len = reverse (tail (take len pieces))
  where pieces = iterate (\ (x, y, z) -> (x+u, y+v, z+w)) start
```

Example:

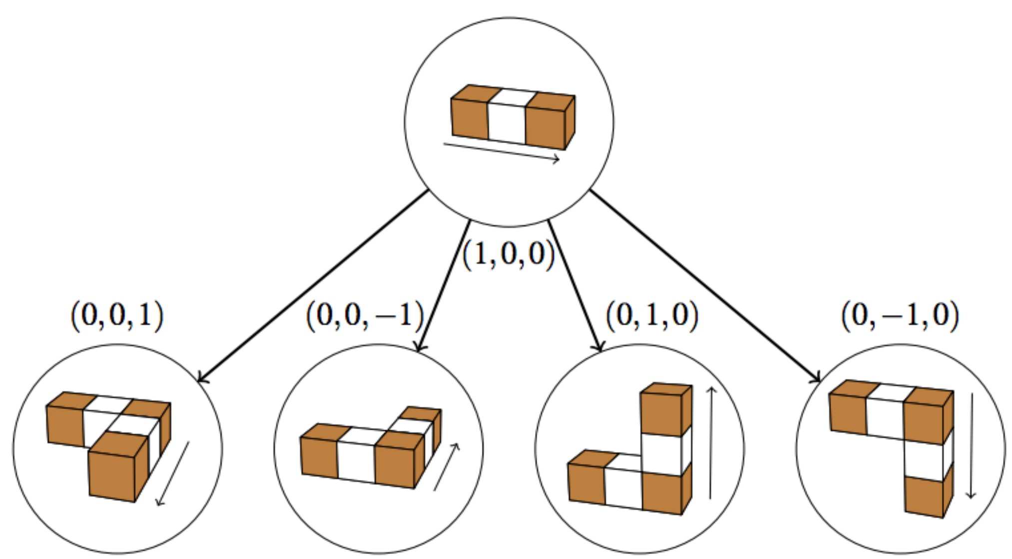
```
section (1, 1, 1) (0, 1, 0) 3 = [(1, 3, 1), (1, 2, 1)]
section (1, 3, 1) (1, 0, 0) 3 = [(3, 3, 1), (2, 3, 1)]
```

Changing Direction



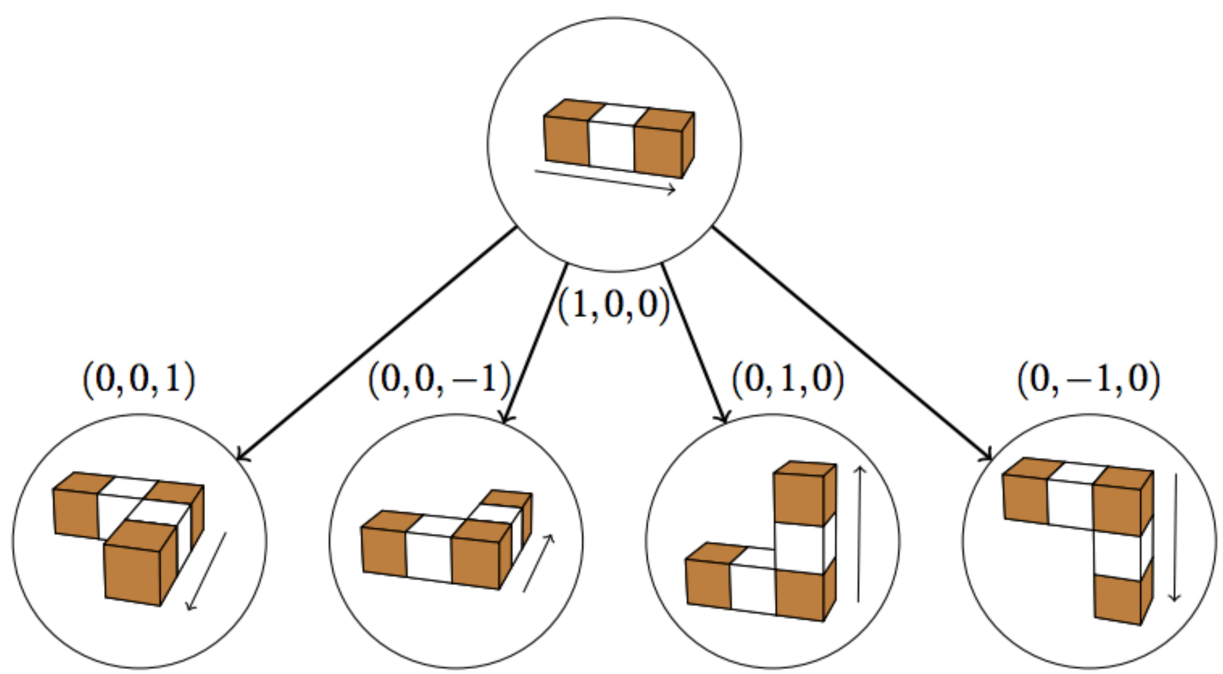
Changing Direction

```
newDirs     :: Direction -> [Direction]
newDirs dir = [ rotate (sign dir) | rotate <- [left,right],
                                    sign   <- [id, inv] ]
 where left, right, inv :: Direction -> Direction
       left  (x, y, z)   = ( y,  z,  x)
       right (x, y, z)   = ( z,  x,  y)
       inv   (x, y, z)   = (-x, -y, -z)
```



Changing Direction

```
newDirs (x, y, z) = [(y, z, x),
                     (-y, -z, -x),
                     (z, x, y),
                     (-z, -x, -y)]
```



Describing Complete Puzzles

```
data Puzzle = Puzzle { sections :: [Int],
                       valid    :: Position -> Bool,
                       initSoln :: Solution,
                       initDir  :: Direction }
```

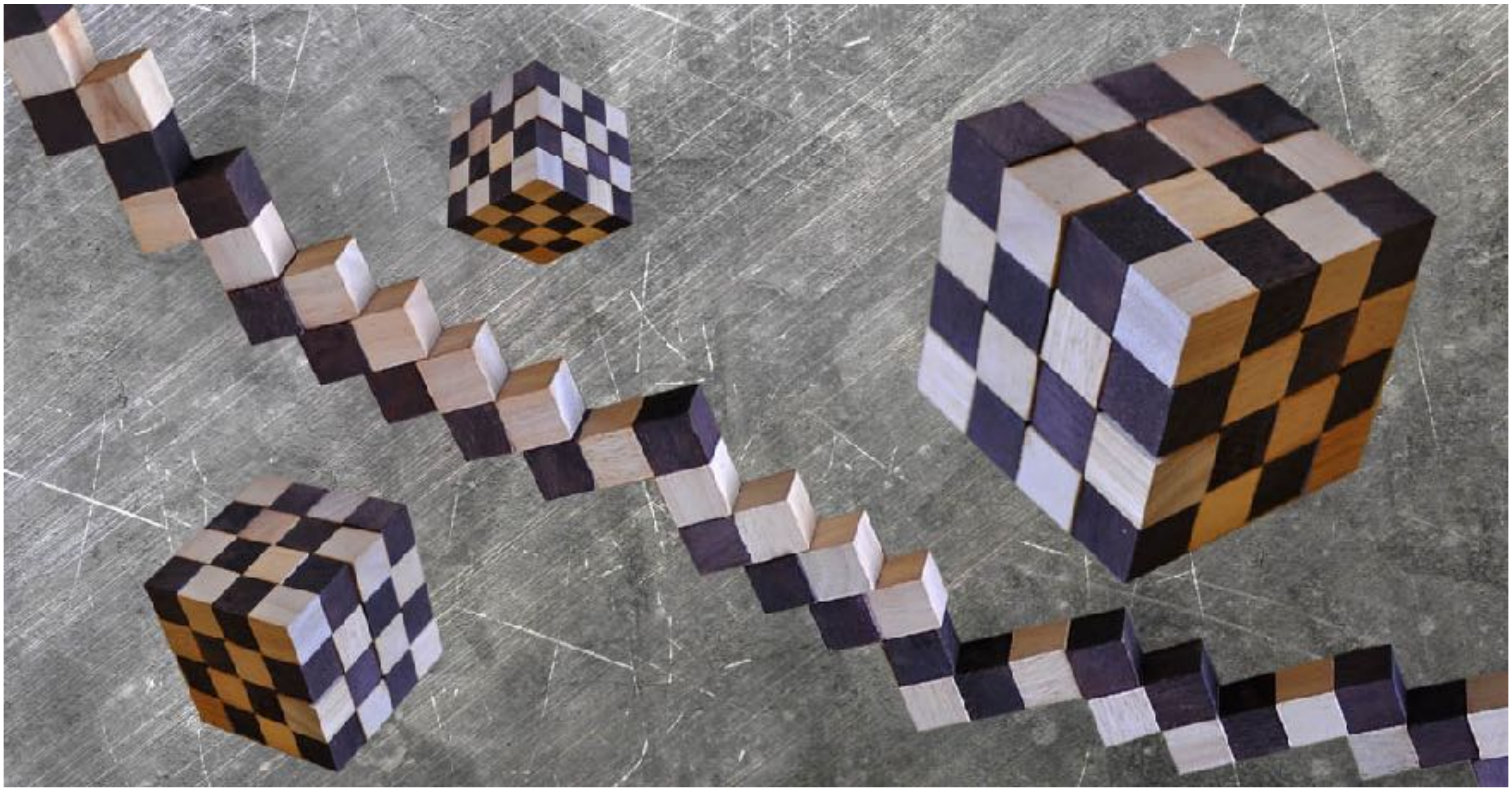
Describing Complete Puzzles

```
standard  :: Puzzle
standard   = Puzzle { valid    = inCube 3,
                      initSoln = [[(1,1,1)]],
                      initDir  = (0,0,1),
                      sections = snake }
```

Describing Complete Puzzles

```
meanGreen :: Puzzle
meanGreen  = standard { sections = [ 3, 3, 2, 3, 2, 3, 2, 2,
                                     2, 3, 3, 3, 2, 3, 3, 3 ] }
```

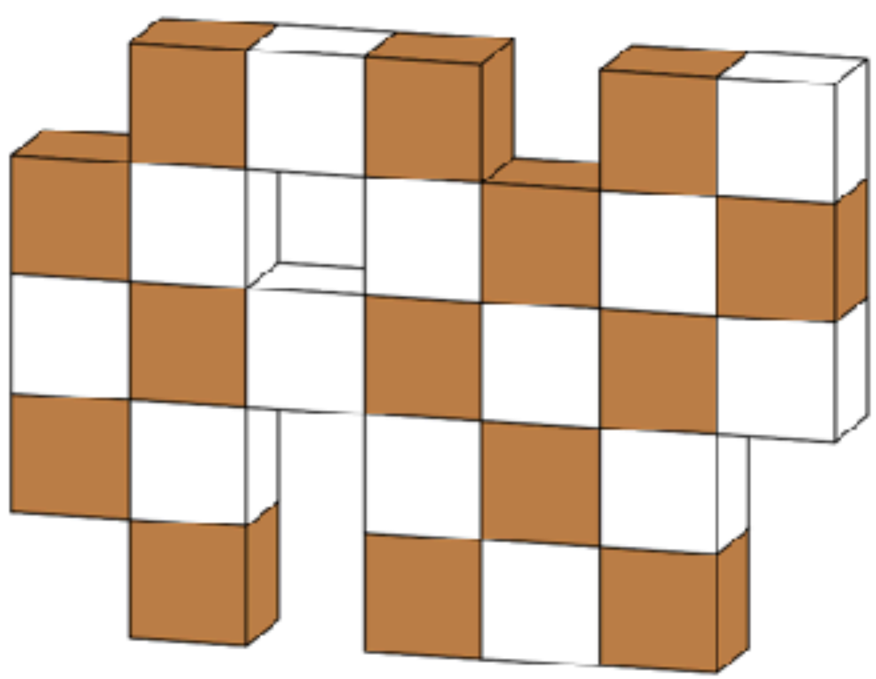
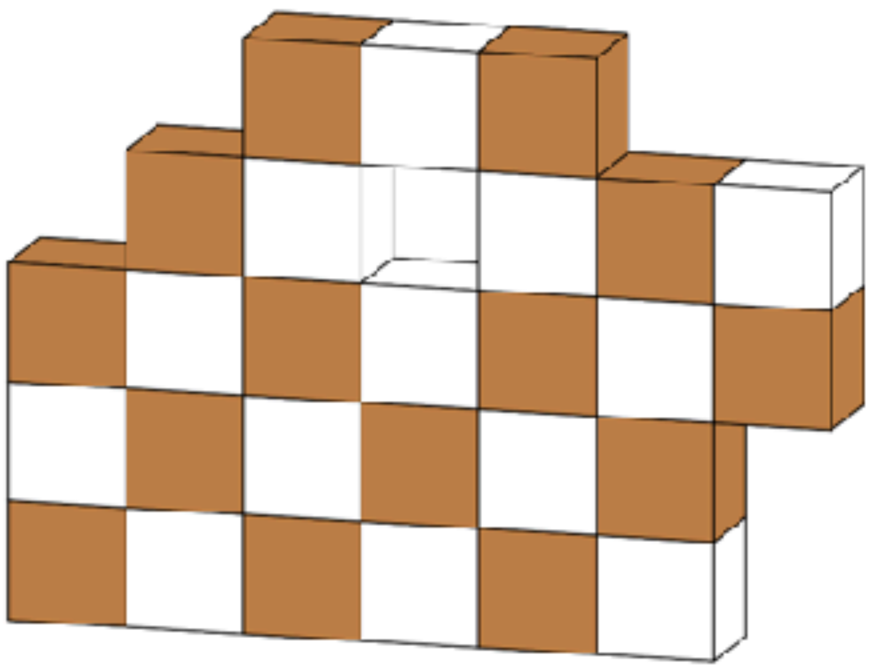
Describing Complete Puzzles



Describing Complete Puzzles

```
king   :: Puzzle
king    = standard {
            valid    = inCube 4,
            sections = [ 3, 2, 3, 2, 2, 4, 2, 3, 2, 3, 2, 3, 2, 2, 2,
                         2, 2, 2, 2, 2, 3, 3, 2, 2, 2, 2, 2, 3, 4, 2,
                         2, 2, 4, 2, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 2 ] }
```

Flat Configurations

```
flat  :: Puzzle -> Puzzle
flat p = p { valid = (\ (x,y,z) -> z==1) }
```

Solving Puzzles

```
extend :: Puzzle -> Solution -> Direction -> Int -> [Solution]
extend p soln dir len = [ sect:soln | let start = head (head soln)
                                          sect  = section start dir len,
                                      all (valid p) sect,
                                      all (`notElem` concat soln) sect ]
```

Replace failure by a list of success!

```
all :: Foldable t => (a -> Bool) -> t a -> Bool
```

Source
#
Determines whether all elements of the structure satisfy the predicate.

Solving Puzzles

```
solutions :: Puzzle -> [Solution]
solutions p = solve (initSoln p) (initDir p) (sections p)
 where solve :: Solution -> Direction -> [Int] -> [Solution]
       solve soln dir [] = [soln]
       solve soln dir (len:lens)
         = concat [ solve soln' newdir lens
                  | newdir <- newDirs dir,
                    soln'  <- extend p soln newdir len ]
```

Solving Puzzles

Alternative

```
solve soln dir (len:lens)
      = do newdir <- newDirs dir
           soln'  <- extend p soln newdir len
           soln'' <- solve soln' newdir lens
           return soln''
```

Solving Puzzles

```
head (solutions standard)
= [[(3, 3, 3), (2, 3, 3)], [(1, 3, 3)],
   [(1, 2, 3)], [(2, 2, 3), (2, 2, 2)],
   [(2, 2, 1)], [(2, 1, 1), (2, 1, 2)],
  [(2, 1, 3)], [(1, 1, 3)], [(1, 1, 2), (1, 2, 2)],
  [(1, 3, 2), (2, 3, 2)], [(3, 3, 2)], [(3, 2, 2)],
  [(3, 2, 3)], [(3, 1, 3), (3, 1, 2)],
  [(3, 1, 1), (3, 2, 1)], [(3, 3, 1), (2, 3, 1)],
  [(1, 3, 1), (1, 2, 1)], [(1, 1, 1)]]
```

The Countdown Problem

Graham Hutton. Journal of Functional Programming, 12(6):609-616, Cambridge University Press, November 2002.

Slides adopted from **http://www.cs.nott.ac.uk/~pszgmh/bib.html#countdown**

What Is 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.

https://www.youtube.com/watch?v=pfa3MHLLSWI

Rules

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.

- We abstract 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 780 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 integers greater than zero 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 an integer greater than zero:

```
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.

Specifying The Problem

Return a list of all possible ways of selecting zero or more elements from a list:

```
subbags :: [a] → [[a]]
```

For example:

```
> subbags [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 → [Int] → Int → Bool
solution e ns n = elem (values e) (subbags ns)
                  && eval e == [n]
```

Brute Force Implementation

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

```
nesplit :: [a] → [([a],[a])]
```

For example:

```
> nesplit [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:

```
exprs      :: [Int] → [Expr]
exprs []   = []
exprs [n]  = [Val n]
exprs ns   = [e | (ls,rs) ← nesplit ns
                , l       ← exprs ls
                , r       ← exprs rs
                , e       ← combine l r]
```

The key function in this lecture.

Combine two expressions using each operator:

```
combine      :: Expr → Expr → [Expr]
combine l r =
   [App o l 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' ← subbags ns
                    , e   ← exprs ns'
                    , eval e == [n]]
```

Correctness Theorem

Our implementation returns all the expressions that satisfy our specification of the problem:

elem e (solutions ns n)

$\Longleftrightarrow$

solution e ns n

How Fast Is It?

1GHz Pentium-III laptop

GHC version 5.00.2

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

0.89 seconds

113.74 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.

Applying Program Fusion

Valid expressions and their values:

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

Specification of a function that fuses together the generation and evaluation of expressions:

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

We can now calculate an implementation:

```
results []  = []
results [n] = [(Val n,n) | n > 0]
results ns  =
   [res | (ls,rs) ← nesplit ns
        , lx      ← results ls
        , ry      ← results rs
        , res     ← combine' lx ry]
```

where

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

Combine’

```
ops                       :: [Op]
ops                        = [Add,Sub,Mul,Div]

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

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

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

Correctness Theorem:

solutions' = solutions

How Fast Is It Now?

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

One solution: 0.08 seconds

All solutions: 5.76 seconds

Can We Do Better?

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

$$x * y \quad = \quad y * x$$

$$x * 1 \quad = \quad x$$

- Exploiting such properties would considerably <u>reduce</u> the search and solution spaces.

Exploiting Properties

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

Using this new predicate gives a new version of our specification of the countdown problem.

Notes:

- The new specification is <u>sound</u> with respect to our original specification.

- It is also <u>complete</u> up to equivalence of expr-essions under the exploited properties.

Using the new predicate also gives a new version of our implementation, written as solutions''.

Notes:

- The new implementation requires no new proof of <u>correctness</u>, because none of our previous proofs depend upon the definition of valid.

- Hence our previous correctness results <u>still hold</u> under changes to this predicate.

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:	0.04 seconds	Around 2 times faster.
All solutions:	0.86 seconds	Around 7 times faster.

More generally, our program usually produces a solution to problems from the television show in an instant, and all solutions in under a second.

Most important points

Program Fusion **Efficiency**

Replace failure by a list of success ! **Elegancy**

Is Carol Now Redundant?

Vorderman said of the computer program:

"It's a nice idea, but thankfully the bosses of Countdown don't want to do away with me and would prefer to keep it human against human!"

How the computer program would handle a makeover show, or dressing up for an awards ceremony remains to be seen.

(London Evening Standard, 1/11/2001)