



Universiteit Utrecht

**[Faculty of Science
Information and Computing Sciences]**

College 2010-2011

5.Types en Lijsten

Doaitse Swierstra

Utrecht University

September 26, 2010

Hammings problem

Hammings problem

Genereer een oplopende lijst van waarden, die geschreven kunnen worden als $\{2^i 3^j 5^k \mid i \geq 0, j \geq 0, k \geq 0\}$.



Hamming's problem

Hamming's problem

Genereer een oplopende lijst van waarden, die geschreven kunnen worden als $\{2^i 3^j 5^k \mid i \geq 0, j \geq 0, k \geq 0\}$.

Een typische manier om dit wat algoritmischer te formuleren is:

1. 1 is een Hamming getal.



Hamming's problem

Hamming's problem

Genereer een oplopende lijst van waarden, die geschreven kunnen worden als $\{2^i 3^j 5^k \mid i \geq 0, j \geq 0, k \geq 0\}$.

Een typische manier om dit wat algoritmischer te formuleren is:

1. 1 is een Hamming getal.
2. Als n een Hamming getal is dat zijn $2 * n$, $3 * n$ en $5 * n$ ook Hamming getallen.



Hamming's problem

Hamming's problem

Genereer een oplopende lijst van waarden, die geschreven kunnen worden als $\{2^i 3^j 5^k \mid i \geq 0, j \geq 0, k \geq 0\}$.

Een typische manier om dit wat algoritmischer te formuleren is:

1. 1 is een Hamming getal.
2. Als n een Hamming getal is dat zijn $2 * n$, $3 * n$ en $5 * n$ ook Hamming getallen.
3. Puriteinen voegen hier aan toe “En er zijn geen andere Hamming getallen”, maar voor Informatici is dit vanzelfsprekend.



Hamming's probleem (code)

We redeneren nu als volgt:

1. Stel dat *ham* de gevraagde lijst is, dan zijn de lijsten *map (*2) ham*, *map (*3) ham*, en *map (*5) ham* ook lijsten met Hamming getallen.



Hamming's probleem (code)

We redeneren nu als volgt:

1. Stel dat *ham* de gevraagde lijst is, dan zijn de lijsten $\text{map } (*2) \text{ ham}$, $\text{map } (*3) \text{ ham}$, en $\text{map } (*5) \text{ ham}$ ook lijsten met Hamming getallen.
2. Als *ham* **monotoon** stijgt dan is dat voor deze nieuwe lijsten ook het geval.



Hamming's probleem (code)

We redeneren nu als volgt:

1. Stel dat *ham* de gevraagde lijst is, dan zijn de lijsten $\text{map } (*2) \text{ ham}$, $\text{map } (*3) \text{ ham}$, en $\text{map } (*5) \text{ ham}$ ook lijsten met Hamming getallen.
2. Als *ham* **monotoon** stijgt dan is dat voor deze nieuwe lijsten ook het geval.
3. Veel getallen zullen in meerdere lijsten voor komen.

| $\text{ham} = 1 : \dots$



Hamming's probleem (code)

We redeneren nu als volgt:

1. Stel dat *ham* de gevraagde lijst is, dan zijn de lijsten $\text{map } (*2) \text{ ham}$, $\text{map } (*3) \text{ ham}$, en $\text{map } (*5) \text{ ham}$ ook lijsten met Hamming getallen.
2. Als *ham* **monotoon** stijgt dan is dat voor deze nieuwe lijsten ook het geval.
3. Veel getallen zullen in meerdere lijsten voor komen.

ham = 1 : ... ($\text{map } (*2) \text{ ham}$)

...

($\text{map } (*3) \text{ ham}$)

...

($\text{map } (*5) \text{ ham}$)



Hamming's probleem (code)

We redeneren nu als volgt:

1. Stel dat *ham* de gevraagde lijst is, dan zijn de lijsten *map (*2) ham*, *map (*3) ham*, en *map (*5) ham* ook lijsten met Hamming getallen.
2. Als *ham* **monotoon** stijgt dan is dat voor deze nieuwe lijsten ook het geval.
3. Veel getallen zullen in meerdere lijsten voor komen.

ham = 1 : ... (*map (*2) ham*)
 '*merge*'
 (*map (*3) ham*)
 '*merge*'
 (*map (*5) ham*)



Hamming's probleem (code)

We redeneren nu als volgt:

1. Stel dat *ham* de gevraagde lijst is, dan zijn de lijsten *map (*2) ham*, *map (*3) ham*, en *map (*5) ham* ook lijsten met Hamming getallen.
2. Als *ham* **monotoon** stijgt dan is dat voor deze nieuwe lijsten ook het geval.
3. Veel getallen zullen in meerdere lijsten voor komen.

```
ham = 1 : remdup ((map (*2) ham)
                  'merge'
                  (map (*3) ham)
                  'merge'
                  (map (*5) ham)
                  )
```

```
remdup (x : ys) = x : remdup (dropWhile ( $\equiv$  x) ys)
```



Vraag

Waarom werkt dit niet met:

$$\begin{aligned} \text{remdup } (x : y : zs) \mid x \equiv y &= \text{remdup } (y : zs) \\ \mid \text{otherwise} &= x : \text{remdup } (y : zs) \end{aligned}$$



Vraag

Waarom werkt dit niet met:

$$\begin{aligned} \text{remdup } (x:y:zs) \mid x \equiv y &= \text{remdup } (y:zs) \\ \mid \text{otherwise} &= x : \text{remdup } (y:zs) \end{aligned}$$

We evalueren een stukje:



Vraag

Waarom werkt dit niet met:

$$\begin{aligned} \text{remdup } (x : y : zs) \mid x \equiv y &= \text{remdup } (y : zs) \\ \mid \text{otherwise} &= x : \text{remdup } (y : zs) \end{aligned}$$
$$\begin{aligned} \text{ham} = 1 : &\text{remdup } ((\text{map } (*2) \text{ ham}) \\ &\quad \text{'merge'} \\ &\quad (\text{map } (*3) \text{ ham}) \\ &\quad \text{'merge'} \\ &\quad (\text{map } (*5) \text{ ham}) \\ &\quad) \end{aligned}$$


Vraag

Waarom werkt dit niet met:

$$\begin{aligned} \text{remdup } (x:y:zs) \mid x \equiv y &= \text{remdup } (y:zs) \\ \mid \text{otherwise} &= x : \text{remdup } (y:zs) \end{aligned}$$
$$\begin{aligned} \text{ham} = & 1 : \text{remdup } ((2 : \text{map } (*2) (\text{tail ham})) \\ & \quad \text{'merge'} \\ & (3 : \text{map } (*3) (\text{tail ham})) \\ & \quad \text{'merge'} \\ & (5 : \text{map } (*5) (\text{tail ham})) \\ &) \end{aligned}$$


Vraag

Waarom werkt dit niet met:

$$\begin{aligned} \text{remdup } (x:y:zs) \mid x \equiv y &= \text{remdup } (y:zs) \\ \mid \text{otherwise} &= x : \text{remdup } (y:zs) \end{aligned}$$
$$\begin{aligned} \text{ham} = & 1 : \text{remdup } ((2 : (\text{map } (*2) (\text{tail ham})) \\ & \quad \quad \quad \text{'merge'} \\ & \quad \quad (3 : \text{map } (*3) (\text{tail ham})) \\ & \quad \quad \text{'merge'} \\ & \quad (5 : \text{map } (*5) (\text{tail ham})) \\ &) \end{aligned}$$


Vraag

Waarom werkt dit niet met:

$$\begin{aligned} \text{remdup } (x:y:zs) \mid x \equiv y &= \text{remdup } (y:zs) \\ \mid \text{otherwise} &= x : \text{remdup } (y:zs) \end{aligned}$$
$$\begin{aligned} \text{ham} = 1 : &\text{remdup } (2 : (\text{map } (*2) (\text{tail ham})) \\ &\quad \text{'merge'} \\ &\quad (3 : \text{map } (*3) (\text{tail ham})) \\ &\quad \text{'merge'} \\ &\quad (5 : \text{map } (*5) (\text{tail ham})) \\ &\quad) \end{aligned}$$


Vraag

Waarom werkt dit niet met:

$$\begin{aligned} \text{remdup } (x : y : zs) \mid x \equiv y &= \text{remdup } (y : zs) \\ \mid \text{otherwise} &= x : \text{remdup } (y : zs) \end{aligned}$$

$$\begin{aligned} \text{ham} = & 1 : \text{remdup } (2 : (((2 * (\text{head } (\text{tail } \text{ham}) : \text{map } (*2) (\text{tail } (\text{tail } \text{ham}))) \\ & \quad \text{'merge'} \\ & \quad (3 : \text{map } (*3) (\text{tail } \text{ham}))) \\ & \quad \text{'merge'} \\ & \quad (5 : \text{map } (*5) (\text{tail } \text{ham}))) \\ & \quad)))) \end{aligned}$$

Voor de *head* (*tail ham*) hebben we het resultaat van *remdup*

nodig!

Universiteit Utrecht



[Faculty of Science
Information and Computing Sciences]

Productiviteit

Vergelijk de twee definities van *remdup*

$$\begin{array}{lcl} \text{remdup } (x : y : zs) & | \ x \equiv y & = \text{remdup } (y : zs) \\ & | \text{ otherwise} & = x : \text{remdup } (y : zs) \\ \text{remdup}' (x : ys) & & = x : \text{remdup}' (\text{dropWhile } (\equiv x) \ ys) \end{array}$$



Productiviteit

Vergelijk de twee definities van *remdup*

$$\begin{aligned} \text{remdup } (x : y : zs) \mid x \equiv y &= \text{remdup } (y : zs) \\ &\mid \text{otherwise} = x : \text{remdup } (y : zs) \\ \text{remdup}' (x : ys) &= x : \text{remdup}' (\text{dropWhile } (\equiv x) \text{ } ys) \end{aligned}$$

Wanneer we dit loslaten op een rij $[1, \langle \text{expr1} \rangle, \langle \text{expr2} \rangle]$ dan evalueert de eerste definitie de $\langle \text{expr1} \rangle$, alvorens de 1 op te leveren. De tweede definitie levert die direct op.



Productiviteit

Vergelijk de twee definities van *remdup*

$$\begin{aligned} \text{remdup } (x : y : zs) \mid x \equiv y &= \text{remdup } (y : zs) \\ &\mid \text{otherwise} = x : \text{remdup } (y : zs) \\ \text{remdup}' (x : ys) &= x : \text{remdup}' (\text{dropWhile } (\equiv x) \text{ } ys) \end{aligned}$$

Wanneer we dit loslaten op een rij $[1, \langle \text{expr1} \rangle, \langle \text{expr2} \rangle]$ dan evalueert de eerste definitie de $\langle \text{expr1} \rangle$, alvorens de 1 op te leveren. De tweede definitie levert die direct op.

Strictness

We zeggen dat de tweede definitie **minder strict** is dan de eerste: als er iets wordt opgeleverd door beide definities is dat hetzelfde, maar de tweede definitie zal vaker (eerder) iets opleveren dan de eerste.



Lijst-comprehensies

Haskell bevat veel syntactische suiker om lijsten gemakkelijk te noteren:

```
? [ (x, y) | x <- [1..5], even x, y <- [1..x]]  
[(2,1),(2,2),(4,1),(4,2),(4,3),(4,4)]
```



Lijst-comprehensies

Haskell bevat veel syntactische suiker om lijsten gemakkelijk te noteren:

```
? [ (x, y) | x <- [1..5], even x, y <- [1..x]]  
[(2,1),(2,2),(4,1),(4,2),(4,3),(4,4)]
```

Anders hadden we moeten schrijven:

```
concat (map f (filter even [1..5]))  
  where f x = map g [1..x]  
         where g y = (x,y)  
concat = foldr (++) []
```



Ook dit is weer syntactische suiker

We noemen de elementen achter de verticale streep **qualifiers**.
Ze hebben een van de volgende drie vormen:



Ook dit is weer syntactische suiker

We noemen de elementen achter de verticale streep qualifiers.
Ze hebben een van de volgende drie vormen:

1. **Generator:** $pat \leftarrow exp$



Ook dit is weer syntactische suiker

We noemen de elementen achter de verticale streep qualifiers.
Ze hebben een van de volgende drie vormen:

1. Generator: $pat \leftarrow exp$
2. Locale declaratie: **let** *decls*



Ook dit is weer syntactische suiker

We noemen de elementen achter de verticale streep qualifiers.
Ze hebben een van de volgende drie vormen:

1. Generator: $pat \leftarrow exp$
2. Locale declaratie: **let** *decls*
3. Guard: *exp*



De vertaling naar "Puur Haskell"

De vertaling van de lijst-comprehensies is in het Haskell rapport gedefiniëerd, met inductie over de lijst van qualifiers:

$$[e \mid \textit{True}] \Rightarrow [e]$$



De vertaling naar "Puur Haskell"

De vertaling van de lijst-comprehensies is in het Haskell rapport gedefiniëerd, met inductie over de lijst van qualifiers:

$$\begin{array}{lcl} [e \mid \textit{True}] & \Rightarrow & [e] \\ [e \mid q] & \Rightarrow & [e \mid q, \textit{True}] \end{array}$$



De vertaling naar "Puur Haskell"

De vertaling van de lijst-comprehensies is in het Haskell rapport gedefiniëerd, met inductie over de lijst van qualifiers:

$$\begin{array}{ll} [e \mid \textit{True}] & \Rightarrow [e] \\ [e \mid q] & \Rightarrow [e \mid q, \textit{True}] \\ [e \mid b, Q] & \Rightarrow \text{if } b \text{ then } [e \mid Q] \text{ else } [] \end{array}$$



De vertaling naar "Puur Haskell"

De vertaling van de lijst-comprehensies is in het Haskell rapport gedefiniëerd, met inductie over de lijst van qualifiers:

$[e \mid \text{True}]$	$\Rightarrow [e]$
$[e \mid q]$	$\Rightarrow [e \mid q, \text{True}]$
$[e \mid b, Q]$	$\Rightarrow \text{if } b \text{ then } [e \mid Q] \text{ else } []$
$[e \mid p \leftarrow l, Q]$	$\Rightarrow \text{let } ok\ p = [e \mid Q]$ $ok\ _ = []$ $\text{in } concat\ (map\ ok\ l)$



De vertaling naar "Puur Haskell"

De vertaling van de lijst-comprehensies is in het Haskell rapport gedefiniëerd, met inductie over de lijst van qualifiers:

$$\begin{aligned} [e \mid \text{True}] &\Rightarrow [e] \\ [e \mid q] &\Rightarrow [e \mid q, \text{True}] \\ [e \mid b, Q] &\Rightarrow \text{if } b \text{ then } [e \mid Q] \text{ else } [] \\ [e \mid p \leftarrow l, Q] &\Rightarrow \text{let } ok\ p = [e \mid Q] \\ &\quad ok\ _ = [] \\ &\quad \text{in } concat\ (map\ ok\ l) \\ [e \mid \text{let } decls, Q] &\Rightarrow \text{let } decls \text{ in } [e \mid Q] \end{aligned}$$



De vertaling naar "Puur Haskell"

De vertaling van de lijst-comprehensies is in het Haskell rapport gedefiniëerd, met inductie over de lijst van qualifiers:

$$\begin{aligned} [e \mid \text{True}] &\Rightarrow [e] \\ [e \mid q] &\Rightarrow [e \mid q, \text{True}] \\ [e \mid b, Q] &\Rightarrow \text{if } b \text{ then } [e \mid Q] \text{ else } [] \\ [e \mid p \leftarrow l, Q] &\Rightarrow \text{let } ok\ p = [e \mid Q] \\ &\quad ok\ _ = [] \\ &\quad \text{in } \text{concat } (\text{map } ok\ l) \\ [e \mid \text{let } decls, Q] &\Rightarrow \text{let } decls \text{ in } [e \mid Q] \end{aligned}$$

Door het patroon links om te schrijven naar het patroon rechts ontstaat stap voor stap een Haskell expressie zonder lijstcomprehensies.



Voorbeeld

We laten in een aantal stappen zien hoe de volgende expressie uitgedrukt kan worden in “simple haskell”.

$$\mathbf{|} \ [(x, y) \mid x \leftarrow [1..5], \text{even } x, y \leftarrow [1..x]]$$



Voorbeeld

| $[(x, y) \mid x \leftarrow [1..5], \text{even } x, y \leftarrow [1..x]]$

De eerste qualifier is een **generator**:

| **let** $ok\ x = [(x, y) \mid \text{even } x, y \leftarrow [1..x]]$
 $ok\ _ = []$

| **in** $concat\ (map\ ok\ [1..5])$



Voorbeeld

```
let ok x = [(x,y) | even x, y ← [1..x]]  
    ok _ = []  
in concat (map ok [1..5])
```

De volgende qualifier is een **guard**:

```
let ok x = if even x  
    then [(x,y) | y ← [1..x]]  
    else []  
    ok _ = []  
in concat (map ok [1..5])
```



Voorbeeld

```
let ok x = if even x
            then [(x,y) | y ← [1..x]]
            else []
ok _ = []
in concat (map ok [1..5])
```

We hebben nu nog maar een enkele qualifier:

```
let ok x = if even x
            then [(x,y) | y ← [1..x], True]
            else []
ok _ = []
in concat (map ok [1..5])
```



Voorbeeld

```
let ok x = if even x
            then [(x,y) | y ← [1..x], True]
            else []
      ok _ = []
in concat (map ok [1..5])
```

We werken de volgende generator weg:

```
let ok x = if even x
            then let ok y = [(x,y) | True]
                  ok _ = []
            in concat (map ok [1..x])
            else []
      ok _ = []
in concat (map ok [1..5])
```



Voorbeeld

```
let ok x = if    even x
            then let ok y = [(x,y) | True]
                  ok _ = []
            in  concat (map ok [1..x])
            else []
ok _ = []
in  concat (map ok [1..5])
```

Vervang de *True*:

```
let ok x = if    even x
            then let ok y = [(x,y)]
                  ok _ = []
            in  concat (map ok [1..x])
            else []
ok _ = []
in  concat (map ok [1..5])
```

Universiteit Utrecht



Voorbeeld

```
let ok x = if    even x
            then let ok y = [(x,y)]
                  ok _ = []
            in  concat (map ok [1..x])
            else []
ok _ = []
in  concat (map ok [1..5])
```

In ons voorbeeld slagen patronen altijd:

```
let ok x = if    even x
            then let ok y = [(x,y)]
                  in  concat (map ok [1..x])
            else []
in  concat (map ok [1..5])
```



Rijttjes getallen

Afkortingen voor rijttjes getallen (*Num*'s) worden als volgt geïnterpreteerd:

$[e1 \dots] \Rightarrow \text{enumFrom } e1$

$[e1, e2 \dots] \Rightarrow \text{enumFromThen } e1 \ e2$

$[e1 \dots e3] \Rightarrow \text{enumFromTo } e1 \ e3$

$[e1, e2 \dots e3] \Rightarrow \text{enumFromThenTo } e1 \ e2 \ e3$

waarbij *enumFrom*, *enumFromThen*, *enumFromTo*, and *enumFromThenTo* gedefinieerd zijn in de prelude.



Rijttjes getallen

Afkortingen voor rijttjes getallen (*Num*'s) worden als volgt geïnterpreteerd:

$[e1 \dots] \Rightarrow \text{enumFrom } e1$
 $[e1, e2 \dots] \Rightarrow \text{enumFromThen } e1 \ e2$
 $[e1 \dots e3] \Rightarrow \text{enumFromTo } e1 \ e3$
 $[e1, e2 \dots e3] \Rightarrow \text{enumFromThenTo } e1 \ e2 \ e3$

waarbij *enumFrom*, *enumFromThen*, *enumFromTo*, and *enumFromThenTo* gedefinieerd zijn in de prelude. Voorbeeld:

```
Programs> [1,4..12]
[1,4,7,10]
Programs> [12,9 ..1]
[12,9,6,3]
```



enumFromThenTo

We moeten bij het ingewikkeldste geval er rekening mee houden dat we zowel “omhoog” als “naar beneden” kunnen aftellen:

```
enumFromByTo :: Int → Int → Int → [Int]  
enumFromByTo x y z = takeWhile stop (iterate next x)  
where stop | x ≤ y = (≤ z)  
        | otherwise = (≥ z)  
        next = (+) (y - x)
```



Tuples

We hebben ze, zonder er heel moeilijk over te doen, al wel eens gebruikt, maar voor de volledigheid vermelden we:

Haskell kent tuples

Voorbeelden van tuples zijn:

- | | |
|----------------------------------|--------------------------------------------------------------------------------------------------------|
| $(1, 'a')$ | een tuple met als elementen de integer 1 en het character 'a'; |
| $(\text{"aap"}, \text{True}, 2)$ | een tuple met drie elementen: de string "aap", de boolean <i>True</i> en het getal 2; |
| $([1,2], \text{sqrt})$ | een tuple met twee elementen: de lijst integers $[1,2]$, en de float-naar-float functie <i>sqrt</i> ; |
| $(1, (2,3))$ | een tuple met twee elementen: het getal 1, en het tuple van de getallen 2 en 3. |



De bijbehorende types zijn

$(1, 'a') :: (Int, Char)$
 $("aap", True, 2) :: ([Char], Bool, Int)$
 $(1, (2, 3)) :: (Int, (Int, Int))$



Functies met meerdere resultaten ..

Worden gesimuleerd met functies die een tuple opleveren:

```
Programs> splitAt 4 "haskell"  
("hask","ell")
```



Functies met meerdere resultaten ..

Worden gesimuleerd met functies die een tuple opleveren:

```
Programs> splitAt 4 "haskell"  
("hask","ell")
```

De bijbehorende code is:

$splitAt :: Int \rightarrow [a] \rightarrow ([a], [a])$
 $splitAt\ n\ xs = (take\ n\ xs, drop\ n\ xs)$



Functies met meerdere resultaten ..

Worden gesimuleerd met functies die een tuple opleveren:

```
Programs> splitAt 4 "haskell"  
("hask", "ell")
```

De bijbehorende code is:

$splitAt :: \text{Int} \rightarrow [a] \rightarrow ([a], [a])$
 $splitAt\ n\ xs = (take\ n\ xs, drop\ n\ xs)$

of beter:

$splitAt\ 0\ xs = ([], xs)$
 $splitAt\ n\ [] = ([], [])$
 $splitAt\ n\ (x : xs) = (x : ys, zs)$
 $\text{where } (ys, zs) = splitAt\ (n - 1)\ xs$



Type Definitions

We geven ingewikkelde types een naam. Dus stel we hebben een stuk code:

$afstand :: (Float, Float) \rightarrow Float$

$verschil :: (Float, Float) \rightarrow (Float, Float) \rightarrow Float$

Dan wordt het vooral bij hogere-orde functies wat lastiger te overzien:

$opp_veelhoek :: [(Float, Float)] \rightarrow Float$

$transf_veelhoek :: ((Float, Float) \rightarrow (Float, Float))$
 $\rightarrow [(Float, Float)] \rightarrow [(Float, Float)]$

In zo'n geval komt een **type-definitie** van pas.



Type Definitions

We geven ingewikkelde types een naam. Met type-definities:

type *Punt* = $(\text{Float}, \text{Float})$

type *veelhoek* = $[\text{Punt}]$

type *Trafo* = $\text{Punt} \rightarrow \text{Punt}$

krijgen we nu:

afstand :: *Punt* \rightarrow *Float*

verschil :: *Punt* \rightarrow *Punt* \rightarrow *Float*

opp_veelhoek :: *veelhoek* \rightarrow *Float*

transf_veelhoek :: *Trafo* \rightarrow *veelhoek* \rightarrow *veelhoek*



Lijstalgoritmen

- ▶ Mooi voorbeeld van combinatorische formuleringen.
- ▶ Komen veel voor.
- ▶ Demonstratie van typische patronen
- ▶ Van nature polymorf.
- ▶ Bruikbaar bij andere vakken.
- ▶ Mooie oefening in het definiëren van recursieve functies.



Combinatorische lijstfuncties

We implementeren de volgende functies:

$$\begin{array}{ll} \text{inits, tails, segs} & :: [a] \rightarrow [[a]] \\ \text{subs, perms} & :: [a] \rightarrow [[a]] \\ \text{combs} & :: \text{Int} \rightarrow [a] \rightarrow [[a]]. \end{array}$$

☞ Al deze functies produceren een lijst van lijsten.



Voorbeelduitvoer

Gegeven de lijst $[1, 2, 3, 4]$:

<i>inits</i>	<i>tails</i>	<i>segs</i>	<i>subs</i>	<i>perms</i>	<i>combs 2</i>	<i>combs 3</i>
$[\]$	$[1, 2, 3, 4]$	$[\]$	$[\]$	$[1, 2, 3, 4]$	$[1, 2]$	$[1, 2, 3]$
$[1]$	$[2, 3, 4]$	$[4]$	$[4]$	$[2, 1, 3, 4]$	$[1, 3]$	$[1, 2, 4]$
$[1, 2]$	$[3, 4]$	$[3]$	$[3]$	$[2, 3, 1, 4]$	$[1, 4]$	$[1, 3, 4]$
$[1, 2, 3]$	$[4]$	$[3, 4]$	$[3, 4]$	$[2, 3, 4, 1]$	$[2, 3]$	$[2, 3, 4]$
$[1, 2, 3, 4]$	$[\]$	$[2]$	$[2]$	$[1, 3, 2, 4]$	$[2, 4]$	
		$[2, 3]$	$[2, 4]$	$[3, 1, 2, 4]$	$[3, 4]$	
		$[2, 3, 4]$	$[2, 3]$	$[3, 2, 1, 4]$		
		$[1]$	$[2, 3, 4]$	$[3, 2, 4, 1]$		
		$[1, 2]$	$[1]$	$[1, 3, 4, 2]$		
		$[1, 2, 3]$	$[1, 4]$	$[3, 1, 4, 2]$		
		$[1, 2, 3, 4]$	$[1, 3]$	$[3, 4, 1, 2]$		
			$[1, 3, 4]$	$[3, 4, 2, 1]$		
			$[1, 2]$	$[1, 2, 4, 3]$		
			$[1, 2, 4]$	$[2, 1, 4, 3]$		
			$[1, 2, 3]$	$[2, 4, 1, 3]$		
			$[1, 2, 3, 4]$	9 andere...		



Alle beginsegmenten

De lege lijst heeft dus één beginsegment: de lege lijst zelf!

| *inits* [] = [[]]



Alle beginsegmenten

De lege lijst heeft dus één beginsegment: de lege lijst zelf!

| $inits [] = [[]]$

Voor het geval $(x : xs)$ kijken we naar de gewenste uitkomsten voor de lijst $[1, 2, 3, 4]$:

| $inits [1, 2, 3, 4] \rightsquigarrow [[], [1], [1, 2], [1, 2, 3], [1, 2, 3, 4]]$

| $inits [\quad 2, 3, 4] \rightsquigarrow [\quad [\quad], [\quad 2], [\quad 2, 3], [\quad 2, 3, 4]].$



Alle beginsegmenten

De lege lijst heeft dus één beginsegment: de lege lijst zelf!

| $inits [] = [[]]$

Voor het geval $(x : xs)$ kijken we naar de gewenste uitkomsten voor de lijst $[1, 2, 3, 4]$:

| $inits [1, 2, 3, 4] \rightsquigarrow [[], [1], [1, 2], [1, 2, 3], [1, 2, 3, 4]]$
 $inits [2, 3, 4] \rightsquigarrow [[], [2], [2, 3], [2, 3, 4]]$.

De elementen van de staart van $inits [1, 2, 3, 4]$ komen overeen met de elementen van $inits [2, 3, 4]$ met een telkens een extra 1 op kop.

Deze lijsten worden aangevuld met de lege lijst.



Alle beginsegmenten

De lege lijst heeft dus één beginsegment: de lege lijst zelf!

| $inits [] = [[]]$

Voor het geval $(x : xs)$ kijken we naar de gewenste uitkomsten voor de lijst $[1, 2, 3, 4]$:

| $inits [1, 2, 3, 4] \rightsquigarrow [[], [1], [1, 2], [1, 2, 3], [1, 2, 3, 4]]$
 $inits [2, 3, 4] \rightsquigarrow [[], [2], [2, 3], [2, 3, 4]]$.

De elementen van de staart van $inits [1, 2, 3, 4]$ komen overeen met de elementen van $inits [2, 3, 4]$ met een telkens een extra 1 op kop.

Deze lijsten worden aangevuld met de lege lijst.

| $inits [] = [[]]$
 $inits (x : xs) = [] : map (x:) (inits xs)$



Alle beginsegmenten: alternatieve definitie

Hoe schrijf je *inits* m.b.v. *foldr*?

$$\text{inits } [] = [[]]$$

$$\text{inits } (x : xs) = [] : \text{map } (x:) (\text{inits } xs)$$



Alle beginsegmenten: alternatieve definitie

Hoe schrijf je *inits* m.b.v. *foldr*?

$$\begin{aligned} \text{inits } [] &= [[]] \\ \text{inits } (x : xs) &= [] : \text{map } (x:) (\text{inits } xs) \end{aligned}$$

$$\text{inits } xs = \text{foldr } (\lambda x r \rightarrow [] : \text{map } (x:) r) [[]] xs$$



Alle beginsegmenten: alternatieve definitie

Hoe schrijf je *inits* m.b.v. *foldr*?

$$\begin{aligned} \text{inits } [] &= [[]] \\ \text{inits } (x : xs) &= [] : \text{map } (x:) (\text{inits } xs) \end{aligned}$$

$$\text{inits } xs = \text{foldr } (\lambda x r \rightarrow [] : \text{map } (x:) r) [[]] xs$$

De parameter r kunnen we gemakkelijk wegwerken door gebruik te maken van functiecompositie, (\circ) :

$$\text{inits } xs = \text{foldr } (\lambda x \rightarrow ([]:) \circ \text{map } (x:)) [[]] xs$$



Alle beginsegmenten: alternatieve definitie

Hoe schrijf je *inits* m.b.v. *foldr*?

$$\begin{aligned} \text{inits } [] &= [[]] \\ \text{inits } (x : xs) &= [] : \text{map } (x:) (\text{inits } xs) \end{aligned}$$

$$\text{inits } xs = \text{foldr } (\lambda x r \rightarrow [] : \text{map } (x:) r) [[]] xs$$

De parameter r kunnen we gemakkelijk wegwerken door gebruik te maken van functiecompositie, (\circ) :

$$\text{inits } xs = \text{foldr } (\lambda x \rightarrow ([]:) \circ \text{map } (x:)) [[]] xs$$

☞ Gebruik van *foldr* maakt niet alles mooier.



Alle beginsegmenten: alternatieve definitie (2)

Het kan nog gekker:

$$\begin{aligned} \text{inits} &= \text{foldr } (\lambda x \ r \rightarrow \ [] : \text{map } (x:) r) \ [] \\ \text{inits} &= \text{foldr } (\lambda x \rightarrow ([]:) \circ \text{map } (x:)) \ [] \\ \text{inits} &= \text{foldr } (\lambda x \rightarrow ([]:) \circ \text{map } ((:) x)) \ [] \\ \text{inits} &= \text{foldr } (\lambda x \rightarrow ([]:) \circ (\text{map} \circ (:)) x) \ [] \\ \text{inits} &= \text{foldr } ((([]) \circ) \circ (\text{map} \circ (:))) \ [] . \end{aligned}$$



Alle beginsegmenten: alternatieve definitie (2)

Het kan nog gekker:

```
inits = foldr (\x r → ( [] :      map  (x:) r ) [[]]
inits = foldr (\x  → ( ( []:) ∘    map  (x:)  ) [[]]
inits = foldr (\x  → ( ( []:) ∘    map  ((:) x)) [[]]
inits = foldr (\x  → ( ( []:) ∘    (map ∘ (:)) x ) [[]]
inits = foldr (      (([]:) ∘) ∘ (map ∘ (:))  ) [[]].
```

Deze stijl van formuleren noemen we **pointfree** of puntvrij.

☞ Alle functies kunnen, *als we zouden willen*, puntvrij, dus zonder lambda geschreven worden.



Alle eindsegmenten

| $\text{tails } [] = [[]]$

Om een idee te krijgen voor de definitie van $\text{tails } (x:xs)$ kijken we eerst weer naar het voorbeeld $[1,2,3,4]$:

| $\text{tails } [1,2,3,4] \rightsquigarrow [[1,2,3,4], [2,3,4], [3,4], [4], []]$
 $\text{tails } [\quad 2,3,4] \rightsquigarrow [\quad \quad [2,3,4], [3,4], [4], []]$



Alle eindsegmenten

| $\text{tails } [] = [[]]$

Om een idee te krijgen voor de definitie van $\text{tails } (x : xs)$ kijken we eerst weer naar het voorbeeld $[1, 2, 3, 4]$:

| $\text{tails } [1, 2, 3, 4] \rightsquigarrow [[1, 2, 3, 4], [2, 3, 4], [3, 4], [4], []]$
 $\text{tails } [\quad 2, 3, 4] \rightsquigarrow [\quad \quad \quad [2, 3, 4], [3, 4], [4], []]$

De elementen van de staart van het resultaat zijn precies gelijk aan de elementen van het resultaat van de recursieve aanroep. Deze lijsten moeten nog aangevuld met de invoerlijst.



Alle eindsegmenten

| $\text{tails } [] = [[]]$

Om een idee te krijgen voor de definitie van $\text{tails } (x : xs)$ kijken we eerst weer naar het voorbeeld $[1, 2, 3, 4]$:

| $\text{tails } [1, 2, 3, 4] \rightsquigarrow [[1, 2, 3, 4], [2, 3, 4], [3, 4], [4], []]$
 $\text{tails } [\quad 2, 3, 4] \rightsquigarrow [\quad \quad [2, 3, 4], [3, 4], [4], []]$

De elementen van de staart van het resultaat zijn precies gelijk aan de elementen van het resultaat van de recursieve aanroep. Deze lijsten moeten nog aangevuld met de invoerlijst.

| $\text{tails } [] = [[]]$
 $\text{tails } (x : xs) = (x : xs) : \text{tails } xs$



Alle eindsegmenten: alternatieve definitie

Kan de functie *tails* weer m.b.v een *foldr* geschreven worden?



Alle eindsegmenten: alternatieve definitie

Kan de functie *tails* weer m.b.v een *foldr* geschreven worden?

Dit is niet zo voor de hand liggend, want we gebruiken behalve de recursieve aanroep *tails xs* ook de waarde *xs* zelf.



Alle eindsegmenten: alternatieve definitie

Kan de functie *tails* weer m.b.v een *foldr* geschreven worden?

Dit is niet zo voor de hand liggend, want we gebruiken behalve de recursieve aanroep *tails xs* ook de waarde *xs* zelf.

Vaak kunnen we dit soort probleemgevallen te lijf gaan door het functieresultaat uit te breiden met een extra component (**tupelen**).



Alle eindsegmenten: alternatieve definitie

Kan de functie *tails* weer m.b.v een *foldr* geschreven worden?

Dit is niet zo voor de hand liggend, want we gebruiken behalve de recursieve aanroep *tails xs* ook de waarde *xs* zelf.

Vaak kunnen we dit soort probleemgevallen te lijf gaan door het functieresultaat uit te breiden met een extra component (tupelen).

In dit geval is de extra component de lijst zelf:

$$\begin{aligned} \text{tls} &= \text{foldr } (\lambda x (xs, r) \rightarrow ((x : xs), (x : xs) : r)) ([], [[]]) \\ \text{tails} &= \text{snd} \circ \text{tls}. \end{aligned}$$


Alle eindsegmenten: alternatieve definitie (2)

Nog “gemakkelijker” is natuurlijk:

$$| \text{ tails } = \text{ reverse } \circ \text{ map reverse } \circ \text{ inits } \circ \text{ reverse }$$



Alle eindsegmenten: alternatieve definitie (2)

Nog “gemakkelijker” is natuurlijk:

$$tails = reverse \circ map\ reverse \circ inits \circ reverse$$

Immers:

$$reverse\ [1..4] \rightsquigarrow [4,3,2,1]$$

$$inits \circ reverse\ \$\ [1..4] \rightsquigarrow [[], [4], [4,3], [4,3,2], [4,3,2,1]]$$

$$map\ reverse \circ inits \circ reverse\ \$\ [1..4] \rightsquigarrow \\ [[], [4], [3,4], [2,3,4], [1,2,3,4]]$$

$$reverse \circ map\ reverse \circ inits \circ reverse\ \$\ [1..4] \rightsquigarrow \\ [[1,2,3,4], [2,3,4], [3,4], [4], []].$$



Alle segmenten

| `segs [] = [[]]`

Voor het geval $(x : xs)$ beginnen we weer met een voorbeeld:

| `segs [1,2,3,4] \rightsquigarrow [[], [4], [3], [3,4], [2], [2,3], [2,3,4],
[1], [1,2], [1,2,3], [1,2,3,4]]`

| `segs [2,3,4] \rightsquigarrow [[], [4], [3], [3,4], [2], [2,3], [2,3,4]].`



Alle segmenten

| $\text{segs } [] = [[]]$

Voor het geval $(x : xs)$ beginnen we weer met een voorbeeld:

| $\text{segs } [1, 2, 3, 4] \rightsquigarrow [[], [4], [3], [3, 4], [2], [2, 3], [2, 3, 4],$
 $[1], [1, 2], [1, 2, 3], [1, 2, 3, 4]]$

| $\text{segs } [2, 3, 4] \rightsquigarrow [[], [4], [3], [3, 4], [2], [2, 3], [2, 3, 4]].$

We kunnen dus schrijven:

| $\text{segs } (x : xs) = \text{segs } xs \mathbin{++} \text{tail } (\text{inits } (x : xs)).$



Alle segmenten

| $\text{segs } [] = [[]]$

Voor het geval $(x : xs)$ beginnen we weer met een voorbeeld:

| $\text{segs } [1, 2, 3, 4] \rightsquigarrow [[], [4], [3], [3, 4], [2], [2, 3], [2, 3, 4],$
 $\quad [1], [1, 2], [1, 2, 3], [1, 2, 3, 4]]$

| $\text{segs } [2, 3, 4] \rightsquigarrow [[], [4], [3], [3, 4], [2], [2, 3], [2, 3, 4]].$

We kunnen dus schrijven:

| $\text{segs } (x : xs) = \text{segs } xs \mathbin{++} \text{tail } (\text{inits } (x : xs)).$

Een andere manier dan gebruik van *tail* om de lege lijst uit de beginsegmenten te verwijderen is

| $\text{segs } [] = [[]]$
| $\text{segs } (x : xs) = \text{segs } xs \mathbin{++} \text{map } (x:) (\text{inits } xs).$



Alle segmenten: probleem

$\text{segs } [] = [[]]$
 $\text{segs } (x : xs) = \text{segs } xs \mathbin{++} \text{map } (x:) (\text{inits } xs)$



Alle segmenten: probleem

```
segs [] = []  
segs (x:xs) = segs xs ++ map (x:) (inits xs)
```

Het op deze manier berekenen van eindsegmenten is vrij duur:



Alle segmenten: probleem

```
segs [] = []  
segs (x:xs) = segs xs ++ map (x:) (inits xs)
```

Het op deze manier berekenen van eindsegmenten is vrij duur:

```
segs (x:y:ys)
```



Alle segmenten: probleem

$$\begin{aligned} \text{segs } [] &= [[]] \\ \text{segs } (x : xs) &= \text{segs } xs \mathrel{++} \text{map } (x:) (\text{inits } xs) \end{aligned}$$

Het op deze manier berekenen van eindsegmenten is vrij duur:

$$\begin{aligned} \text{segs } (x : y : ys) \\ &= \\ \text{segs } (\quad y : ys) &\quad \mathrel{++} \text{map } (x:) (\text{inits } (y : ys)) \end{aligned}$$



Alle segmenten: probleem

$$\begin{aligned} \text{segs } [] &= [[]] \\ \text{segs } (x : xs) &= \text{segs } xs \mathbin{++} \text{map } (x:) (\text{inits } xs) \end{aligned}$$

Het op deze manier berekenen van eindsegmenten is vrij duur:

$$\begin{aligned} &\text{segs } (x : y : ys) \\ &= \\ &\text{segs } (\quad y : ys) \qquad \qquad \qquad \mathbin{++} \text{map } (x:) (\text{inits } (y : ys)) \\ &= \\ &\text{segs } \qquad \qquad \qquad ys \mathbin{++} \text{map } (y:) (\text{inits } ys) \mathbin{++} \text{map } (x:) (\text{inits } (y : ys)) \end{aligned}$$



Alle segmenten: probleem

$$\begin{aligned} \text{segs } [] &= [[]] \\ \text{segs } (x : xs) &= \text{segs } xs \mathbin{++} \text{map } (x:) (\text{inits } xs) \end{aligned}$$

Het op deze manier berekenen van eindsegmenten is vrij duur:

$$\begin{aligned} &\text{segs } (x : y : ys) \\ &= \\ &\text{segs } (\quad y : ys) \quad \mathbin{++} \text{map } (x:) (\text{inits } (y : ys)) \\ &= \\ &\text{segs } \quad ys \mathbin{++} \text{map } (y:) (\text{inits } ys) \mathbin{++} \text{map } (x:) (\text{inits } (y : ys)) \\ &= \\ &\text{segs } \quad ys \mathbin{++} \text{map } (y:) (\text{inits } ys) \mathbin{++} \\ &\quad \text{map } (x:) ([] : \text{map } (y:) : \text{inits } ys)). \end{aligned}$$



Alle segmenten: probleem

$$\begin{aligned} \text{segs } [] &= [[]] \\ \text{segs } (x : xs) &= \text{segs } xs \mathbin{++} \text{map } (x:) (\text{inits } xs) \end{aligned}$$

Het op deze manier berekenen van eindsegmenten is vrij duur:

$$\begin{aligned} &\text{segs } (x : y : ys) \\ &= \\ &\text{segs } (\quad y : ys) \quad \mathbin{++} \text{map } (x:) (\text{inits } (y : ys)) \\ &= \\ &\text{segs } \quad ys \mathbin{++} \text{map } (y:) (\text{inits } ys) \mathbin{++} \text{map } (x:) (\text{inits } (y : ys)) \\ &= \\ &\text{segs } \quad ys \mathbin{++} \text{map } (y:) (\text{inits } ys) \mathbin{++} \\ &\quad \text{map } (x:) ([] : \text{map } (y:) (\text{inits } ys)). \end{aligned}$$



Alle segmenten: oplossing

Een oplossing voor dit soort problemen wordt weer gevonden in het **tupelen** (tegelijk uitvoeren) van beide berekeningen:

```
inits []           = [[]]
inits (x : xs)     = [] : map (x:) (inits xs)

segs []           = [[]]
segs (x : xs)     = segs xs ++ map (x:) (inits xs)

segsinits []       = ([[]], [[]])
segsinits (x : xs) = let (ys, zs) = segsinits xs
                      in (ys ++ map (x:) zs
                          , [] : map (x:) zs
                          )

fastsegs           = fst ∘ segsinits.
```



je ziet het verschil

Door in de GHC de optie `:set +s` aan te zetten, krijg je de gebruikte tijd te zien:

```
*Programs> :set +s
*Programs> length (fastsegs [1..50])
1276
(0.03 secs, 1067580 bytes)
*Programs> length (segs [1..50])
1276
(0.31 secs, 1936536 bytes)
```



je ziet het verschil

Door in de GHC de optie `:set +s` aan te zetten, krijg je de gebruikte tijd te zien:

```
*Programs> :set +s
*Programs> length (fastsegs [1..50])
1276
(0.03 secs, 1067580 bytes)
*Programs> length (segs [1..50])
1276
(0.31 secs, 1936536 bytes)
```

Er word veel onderzoek gedaan naar hoe een compiler zulk soort optimalisaties ook zelf kan doen.



Alle segmenten: alternatieve definitie

$$\text{segs } xs = [] : [t \mid i \leftarrow \text{inits } xs, t \leftarrow \text{tails } i, \neg (\text{null } t)]$$



Alle segmenten: alternatieve definitie

$\text{segs } xs = [] : [t \mid i \leftarrow \text{inits } xs, t \leftarrow \text{tails } i, \neg (\text{null } t)]$

```
*Main> length (segs [1 .. 50])  
1276  
(0.10 secs, 2467236 bytes)
```



Alle sublijsten

$\text{subs } [1, 2, 3, 4] \rightsquigarrow [[1, 2, 3, 4], [1, 2, 3], [1, 2, 4], [1, 2],$
[1, 3, 4], [1, 3], [1, 4], [1],
[2, 3, 4], [2, 3], [2, 4], [2],
[3, 4], [3], [4], []]

$\text{subs } [2, 3, 4] \rightsquigarrow [[2, 3, 4], [2, 3], [2, 4], [2], [3, 4], [3], [4], []]$



Alle sublijsten

```
subs [1,2,3,4] ~> [[1,2,3,4], [1,2,3], [1,2,4], [1,2],  
                  [1,3,4], [1,3], [1,4], [1],  
                  [2,3,4], [2,3], [2,4], [2],  
                  [3,4], [3], [4], []]  
subs [2,3,4]   ~> [[2,3,4], [2,3], [2,4], [2], [3,4], [3], [4], []]
```

De definitie kan dus luiden:

```
subs []      = [[]]  
subs (x:xs) = map (x:) (subs xs) ++ subs xs.
```



Alle sublijsten

```
subs [1,2,3,4] ~> [[1,2,3,4], [1,2,3], [1,2,4], [1,2],  
                  [1,3,4], [1,3], [1,4], [1],  
                  [2,3,4], [2,3], [2,4], [2],  
                  [3,4], [3], [4], []]  
subs [2,3,4]   ~> [[2,3,4], [2,3], [2,4], [2], [3,4], [3], [4], []]
```

De definitie kan dus luiden:

```
subs []      = [[]]  
subs (x:xs) = map (x:) (subs xs) ++ subs xs.
```

Of beter:

```
subs []      = [[]]  
subs (x:xs) = map (x:) ys ++ ys  
              where ys = subs xs.
```



Alle permutaties

perms $[1, 2, 3, 4]$ \rightsquigarrow

$[1, 2, 3, 4], [2, 1, 3, 4], [2, 3, 1, 4], [2, 3, 4, 1],$
 $[1, 3, 2, 4], [3, 1, 2, 4], [3, 2, 1, 4], [3, 2, 4, 1],$
 $[1, 3, 4, 2], [3, 1, 4, 2], [3, 4, 1, 2], [3, 4, 2, 1],$
 $[1, 2, 4, 3], [2, 1, 4, 3], [2, 4, 1, 3], [2, 4, 3, 1],$
 $[1, 4, 2, 3], [4, 1, 2, 3], [4, 2, 1, 3], [4, 2, 3, 1],$
 $[1, 4, 3, 2], [4, 1, 3, 2], [4, 3, 1, 2], [4, 3, 2, 1]$

perms $[2, 3, 4]$ \rightsquigarrow

$[2, 3, 4], [3, 2, 4], [3, 4, 2], [2, 4, 3], [4, 2, 3], [4, 3, 2]$



Alle permutaties: definitie

Voeg een element toe op alle mogelijke plaatsen:

between $:: a \rightarrow [a] \rightarrow [[a]]$

between $e [] = [[e]]$

between $e (y:ys) = (e:y:ys) : \text{map } (y:) (\text{between } e \text{ } ys)$




Alle permutaties: definitie

Voeg een element toe op alle mogelijke plaatsen:

$$\begin{aligned} \text{between} &:: a \rightarrow [a] \rightarrow [[a]] \\ \text{between } e \ [] &= [[e]] \\ \text{between } e \ (y:ys) &= (e:y:ys) : \text{map } (y:) (\text{between } e \ ys) \end{aligned}$$

Of met een accumulerende parameter \wp , die het al doorlopen stuk representeert:


$$\begin{aligned} \text{between} &= \text{btwn } \text{id} \\ \text{where} \\ \text{btwn } \wp \ e \ [] &= \wp \ e \\ \text{btwn } \wp \ e \ (x:xs) &= \wp \ (e:x:xs) : \text{btwn } (\wp \circ (x:)) \ e \ xs. \end{aligned}$$



Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

32



Voeg een element toe op alle mogelijke plaatsen:

between $\quad \quad \quad \vdash a \rightarrow [a] \rightarrow [[a]]$

between $e \text{ []} = [[e]]$

between e $(y:ys) = (e:y:ys) : \text{map } (y:.)$ (between e ys)

Of met een accumulerende parameter \wp , die het al doorlopen stuk representeert:

between = *btwn* id

where

$$btwn \wp e \left[\begin{array}{c} \text{ } \end{array} \right] = \wp e$$
$$btwn \wp e (x:xs) = \wp (e:x:xs) : btwn (\wp \circ (x:)) e xs.$$


Alle permutaties: definitie

Voeg een element toe op alle mogelijke plaatsen:

$between \quad \quad \quad :: a \rightarrow [a] \rightarrow [[a]]$
 $between\ e\ [] \quad \quad = [[e]]$
 $between\ e\ (y:ys) = (e:y:ys) : map\ (y:) \ (between\ e\ ys)$

Of met een accumulerende parameter \wp , die het al doorlopen stuk representeert:

$between = btwn\ id$
where
 $btwn\ \wp\ e\ [] \quad \quad = \wp\ e$
 $btwn\ \wp\ e\ (x:xs) = \wp\ (e:x:xs) : btwn\ (\wp \circ (x:))\ e\ xs.$

$perms\ [] \quad \quad = [[]]$
 $perms\ (x:xs) = concat\ (map\ (between\ x)\ (perms\ xs))$



Alle permutaties: lexicografische volgorde

Om de permutaties van een gesorteerde lijst in lexicografische volgorde op te sommen, berekenen we eerst alle paren bestaande uit een element van de lijst en de overige elementen:

```
*Main> splits id [1, 2, 3]  
[(1,[2,3]),(2,[1,3]),(3,[1,2])]
```



Alle permutaties: lexicografische volgorde

Om de permutaties van een gesorteerde lijst in lexicografische volgorde op te sommen, berekenen we eerst alle paren bestaande uit een element van de lijst en de overige elementen:

$$\begin{aligned} \text{plits} &:: ([a] \rightarrow [a]) \rightarrow [a] \rightarrow [(a, [a])] \\ \text{plits} \quad \emptyset [] &= [] \\ \text{plits} \quad \emptyset (x : xs) &= (x, \emptyset xs) : \\ &\quad \text{plits} (\emptyset \circ (x:)) xs. \end{aligned}$$



Alle permutaties: lexicografische volgorde

Om de permutaties van een gesorteerde lijst in lexicografische volgorde op te sommen, berekenen we eerst alle paren bestaande uit een element van de lijst en de overige elementen:

$$\begin{aligned} \text{plits} &:: ([a] \rightarrow [a]) \rightarrow [a] \rightarrow [(a, [a])] \\ \text{plits} \quad \emptyset [] &= [] \\ \text{plits} \quad \emptyset (x : xs) &= (x, \emptyset xs) : \\ &\quad \text{plits} (\emptyset \circ (x:)) xs. \end{aligned}$$

De geordende permutaties laten zich dan berekenen met

$$\begin{aligned} \text{perms} [] &= [[]] \\ \text{perms } xs &= [y : zs \mid (y, ys) \leftarrow \text{plits id } xs, zs \leftarrow \text{perms } ys]. \end{aligned}$$


transpose

Een veelgebruikte functie is *transpose* :: $[[a]] \rightarrow [[a]]$:

```
transpose [[1,2, 3]
           ,      [4,5, 6]
           ,      [7,8, 9]
           ,      [10,11,12]
           ] = [[1,4,7,10]
                , [2,5,8,11]
                , [3,6,9,12]
                ]
```



transpose

Een veelgebruikte functie is *transpose* :: $[[a]] \rightarrow [[a]]$:

```
transpose [[1,2, 3]
           ,      [4,5, 6]
           ,      [7,8, 9]
           ,      [10,11,12]
           ] = [[1,4,7,10]
                , [2,5,8,11]
                , [3,6,9,12]
                ]
```

Of een beetje suggestief in een plaatje:

```
[[1, 2, 3 ]   [[1,4,7,10]
 , [4, 5, 6 ] , [2,5,8,11]
 , [7, 8, 9 ]  $\Rightarrow$  , [3,6,9,12]
 , [10,11,12] ]
]
```

Universiteit Utrecht



transpose ...

Bekijk eens:

$$\left[\begin{array}{l} [[\dots, \dots, \dots] \\ , [4, 5, 6] \\ , [7, 8, 9] \\ , [10, 11, 12] \end{array} \right] \Rightarrow \left[\begin{array}{l} [[\dots, 4, 7, 10] \\ , [\dots, 5, 8, 11] \\ , [\dots, 6, 9, 12] \end{array} \right]$$



transpose ...

Bekijk eens:

$$\begin{bmatrix} [\dots, \dots, \dots] \\ [4, 5, 6] \\ [7, 8, 9] \\ [10, 11, 12] \end{bmatrix} \Rightarrow \begin{bmatrix} [\dots, 4, 7, 10] \\ [\dots, 5, 8, 11] \\ [\dots, 6, 9, 12] \end{bmatrix}$$

Dit geeft ons direct inspiratie voor een recursieve formulering:



transpose ...

Bekijk eens:

$$\begin{bmatrix} [\dots, \dots, \dots] \\ [4, 5, 6] \\ [7, 8, 9] \\ [10, 11, 12] \end{bmatrix} \Rightarrow \begin{bmatrix} [\dots, 4, 7, 10] \\ [\dots, 5, 8, 11] \\ [\dots, 6, 9, 12] \end{bmatrix}$$

Dit geeft ons direct inspiratie voor een recursieve formulering:

$$\text{transpose } (x : xs) = \dots \text{transpose } xs$$



transpose ...

Bekijk eens:

$$\begin{bmatrix} [\dots, \dots, \dots] \\ [4, 5, 6] \\ [7, 8, 9] \\ [10, 11, 12] \end{bmatrix} \Rightarrow \begin{bmatrix} [\dots, 4, 7, 10] \\ [\dots, 5, 8, 11] \\ [\dots, 6, 9, 12] \end{bmatrix}$$

Dit geeft ons direct inspiratie voor een recursieve formulering:

$$\text{transpose } (x : xs) = \text{zipWith } (:) x (\text{transpose } xs)$$


transpose ...

Bekijk eens:

$$\begin{bmatrix} [\dots, \dots, \dots] \\ [4, 5, 6] \\ [7, 8, 9] \\ [10, 11, 12] \end{bmatrix} \Rightarrow \begin{bmatrix} [\dots, 4, 7, 10] \\ [\dots, 5, 8, 11] \\ [\dots, 6, 9, 12] \end{bmatrix}$$

Dit geeft ons direct inspiratie voor een recursieve formulering:

$$\text{transpose } xs = \text{foldr } (\text{zipWith } (:)) \text{ ??? } xs$$


transpose ...

Bekijk eens:

```
[ [..., ..., ...]  [..., 4, 7, 10]  
  , [4,  5,  6 ]    , [..., 5, 8, 11]  
  , [7,  8,  9 ]    ⇒ , [..., 6, 9, 12]  
  , [10, 11, 12]    ]  
  ]
```

Dit geeft ons direct inspiratie voor een recursieve formulering:

```
transpose xs = foldr (zipWith (:)) (repeat []) xs
```

