Universiteit Utrecht

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

# College 201-2012
# 12. Rekenen met Programma's

Doaitse Swierstra

Utrecht University

October 25, 2011

# Aspecten die het tijd/ruimte gedrag beïnvloeden

Het is niet altijd eenvoudig direct in te zien hoveel tijd en ruimte een Haskell programma vraagt:

1. expressies worden gedeeld

# Aspecten die het tijd/ruimte gedrag beïnvloeden

Het is niet altijd eenvoudig direct in te zien hoveel tijd en ruimte een Haskell programma vraagt:

1. expressies worden gedeeld
2. expressies worden pas uitgerekend als de waarde nodig is

# Aspecten die het tijd/ruimte gedrag beïnvloeden

Het is niet altijd eenvoudig direct in te zien hoveel tijd en ruimte een Haskell programma vraagt:

1. expressies worden gedeeld
2. expressies worden pas uitgerekend als de waarde nodig is
3. expressies worden laat uitgerekend, maar wel opgebouwd

# Aspecten die het tijd/ruimte gedrag beïnvloeden

Het is niet altijd eenvoudig direct in te zien hoveel tijd en ruimte een Haskell programma vraagt:

1. expressies worden gedeeld
2. expressies worden pas uitgerekend als de waarde nodig is
3. expressies worden laat uitgerekend, maar wel opgebouwd
4. referenties naar waarden die je niet meer gaat gebruiken blijven "hangen"

# Wat kunnen we daar aan doen?

Er is niet één oplossing voor al deze problemen, maar

1. gebruik bibliotheken en de daarin gedefiniëerde types; vaak ben je niet de eerste die tegen het probleem aanloopt

# Wat kunnen we daar aan doen?

Er is niet één oplossing voor al deze problemen, maar

1. gebruik bibliotheken en de daarin gedefiniëerde types; vaak ben je niet de eerste die tegen het probleem aanloopt
2. vertrouw er op dat de vertaler ook vaak al slim is

# Wat kunnen we daar aan doen?

Er is niet één oplossing voor al deze problemen, maar

1. gebruik bibliotheken en de daarin gedefiniëerde types; vaak ben je niet de eerste die tegen het probleem aanloopt
2. vertrouw er op dat de vertaler ook vaak al slim is
3. gebruik strictheidsannotaties, gebruik “unboxed” waarden

# Wat kunnen we daar aan doen?

Er is niet één oplossing voor al deze problemen, maar

1. gebruik bibliotheken en de daarin gedefiniëerde types; vaak ben je niet de eerste die tegen het probleem aanloopt
2. vertrouw er op dat de vertaler ook vaak al slim is
3. gebruik strictheidsannotaties, gebruik "unboxed" waarden
4. bouw zelf een verzameling patronen op, en kijk op `www.haskell.org/haskellwiki/Category:Idioms`

# Wat kunnen we daar aan doen?

Er is niet één oplossing voor al deze problemen, maar

1. gebruik bibliotheken en de daarin gedefiniëerde types; vaak ben je niet de eerste die tegen het probleem aanloopt
2. vertrouw er op dat de vertaler ook vaak al slim is
3. gebruik strictheidsannotaties, gebruik "unboxed" waarden
4. bouw zelf een verzameling patronen op, en kijk op `www.haskell.org/haskellwiki/Category:Idioms`
5. ga m.b.v. heap en tijd-profilering na waar de ruimte en de tijd gaat zitten, en transformeer je programma zo mogelijk naar een efficiëntere versie, of kijk op `http://www.haskell.org/haskellwiki/Performance`

# Delen van waarden

De functie $g$ gebruikt $k$:

$$
\begin{aligned}
f\ x &= product\ [1\,..\,x] \\
k &= f\ 5 \\
g\ x &= k + 1
\end{aligned}
$$

# Delen van waarden

De functie $g$ gebruikt $k$:

$$\begin{aligned} f\ x &= product\ [1\,..\,x] \\ k &= f\ 5 \\ g\ x &= k + 1 \end{aligned}$$

De constante $k$ wordt dankzij lazy evaluatie pas uitgerekend als hij voor het eerst gebruikt wordt, bijvoorbeeld door een aanroep van $g$:

```
? g 1
121
(57 reductions, 78 cells)
```

# Delen van waarden

De functie $g$ gebruikt $k$:

$$\begin{aligned} f\ x &= product\ [1\,..\,x] \\ k &= f\ 5 \\ g\ x &= k + 1 \end{aligned}$$

De constante $k$ wordt dankzij lazy evaluatie pas uitgerekend als hij voor het eerst gebruikt wordt, bijvoorbeeld door een aanroep van $g$:

```
? g 1
121
(57 reductions, 78 cells)
```

De waarde van $k$ is nu bekend:

```
? g 1
121
(3 reductions, 8 cells)
```

# Kosten van partiële parametrisatie

Functies die zijn gedefinieerd door een hogere-orde functie partieel te parametriseren, bijvoorbeeld

$$som = foldr\ (+)\ 0$$

kosten de tweede keer dat ze gebruikt worden één reductie minder:

```
? som [1,2,3]
6
(9 reductions, 19 cells)
? som [1,2,3]
6
(8 reductions, 17 cells)
```

# Veel voorkomende complexiteiten

Bij de analyse van het tijdgebruik van een functie is vaak niet het precieze aantal reducties van belang. Veel interessanter is het om te zien hoe dit aantal reducties verandert als de grootte van de invoer verandert.

| elementen | *head* | *last* | *length* | *sum* | $(+\!\!+x)$ | *isort* |
|---:|---:|---:|---:|---:|---:|---:|
| 10 | 2 | 11 | 42 | 32 | 52 | 133 |
| 100 | 2 | 101 | 402 | 302 | 502 | 10303 |
| 1000 | 2 | 1001 | 4002 | 3002 | 5002 | 1003003 |
| $n$ | 2 | $n+1$ | $4n+2$ | $3n+2$ | $5n+2$ | $n^2+3n+3$ |

# Hoe snel neemt de rekentijd toe bij ...

De volgende tabel geeft enig gevoel voor de snelheid waarmee de verschillende functies stijgen:

| orde | $n = 10$ | $n = 100$ | $n = 1000$ | naam |
|---|---|---|---|---|
| $\mathcal{O}(1)$ | 1 | 1 | 1 | constant |
| $\mathcal{O}(\log n)$ | 3 | 7 | 10 | logaritmisch |
| $\mathcal{O}(\sqrt{n})$ | 3 | 10 | 32 | |
| $\mathcal{O}(n)$ | 10 | 100 | 1000 | lineair |
| $\mathcal{O}(n \log n)$ | 30 | 700 | 10000 | |
| $\mathcal{O}(n\sqrt{n})$ | 30 | 1000 | 31623 | |
| $\mathcal{O}(n^2)$ | 100 | 10000 | $10^6$ | kwadratisch |
| $\mathcal{O}(n^3)$ | 1000 | $10^6$ | $10^9$ | kubisch |
| $\mathcal{O}(2^n)$ | 1024 | $10^{30}$ | $10^{300}$ | exponentieel |

# Typische gevallen

- De functie is niet recursief, en bevat alleen aanroepen van functies met complexiteit $\mathcal{O}(1)$. Deze functies hebben zelf ook complexiteit $\mathcal{O}(1)$.
- De functie doet één recursieve aanroep van zichzelf met een iets kleinere parameter.
  Deze functies hebben complexiteit $\mathcal{O}(n)$.
- De functie doet één recursieve aanroep van zichzelf met een iets kleinere parameter. Het eindantwoord wordt vervolgens in lineaire tijd bepaald. Deze functies hebben complexiteit $\mathcal{O}(n^2)$.
- De functie doet *twee* recursieve aanroepen van zichzelf met iets kleinere parameter. Deze functies hebben complexiteit $\mathcal{O}(2^n)$.

# ... continued ...

- De functie doet één recursieve aanroep van zichzelf met een parameter die ongeveer twee keer zo klein geworden is. Het eindantwoord wordt vervolgens in constante tijd berekend. eze functies hebben complexiteit $\mathcal{O}(\log n)$.
- De functie deelt zijn parameter ongeveer in tweeën en roept zichzelf recursief aan op *beide* delen. Deze functies hebben complexiteit $\mathcal{O}(n)$.
- De functie deelt zijn parameter ongeveer in tweeën en roept zichzelf recursief aan op beide delen. Het eindantwoord wordt vervolgens in lineaire tijd berekend. Deze functies hebben complexiteit $\mathcal{O}(n \log n)$.

# In een tabel:

| parameter | aant.rec.aanr. | vervolgens | voorbeeld | complexitei |
|---|---|---|---|---|
| geen | 0 | constant | *head* | $\mathcal{O}(1)$ |
| kleiner | 1 | constant | *sum* | $\mathcal{O}(n)$ |
| kleiner | 1 | lineair | *isort* | $\mathcal{O}(n^2)$ |
| kleiner | 2 | will. | *subs* | $\mathcal{O}(2^n)$ |
| helft | 1 | constant | *elemBoom* | $\mathcal{O}(\log n)$ |
| helft | 1 | lineair | | $\mathcal{O}(n)$ |
| helft | 2 | constant | *omvang* | $\mathcal{O}(n)$ |
| helft | 2 | lineair | *msort* | $\mathcal{O}(n \log n)$ |

# Kleine parameters

De operator $+\!\!+$ is gedefinieerd met inductie naar de linker parameter:

$$
\begin{aligned}
[\,] \quad\quad\quad &+\!\!+\, ys = ys \\
(x : xs) &+\!\!+\, ys = x : (xs +\!\!+\, ys)
\end{aligned}
$$

Als twee lijsten samengevoegd moeten worden, maar de volgorde van de elementen doet er niet toe, dan is het dus efficiënter om de kortste van de twee voorop te zetten.

# Kleine parameters

De operator $+\!\!+$ is gedefinieerd met inductie naar de linker parameter:

$$
\begin{aligned}
[\,] \quad\quad\quad &+\!\!+\, ys = ys \\
(x : xs) &+\!\!+\, ys = x : (xs +\!\!+\, ys)
\end{aligned}
$$

Als twee lijsten samengevoegd moeten worden, maar de volgorde van de elementen doet er niet toe, dan is het dus efficiënter om de kortste van de twee voorop te zetten.

Stel dat de lengte van $xs$ gelijk is aan $x$, de lengte van $ys$ $y$ en die van $zs$ $z$. Dan geldt:

| *tijd voor...* | $(xs +\!\!+\, ys) +\!\!+\, zs$ | $xs +\!\!+\, (ys +\!\!+\, zs)$ |
|---|---|---|
| *eerste* $+\!\!+$ | $x$ | $x$ |
| *tweede* $+\!\!+$ | $x + y$ | $y$ |
| *totaal* | $2x + y$ | $x + y$ |

# *ShowS*

Vaak kunnen we ook werk uitstellen totdat we alle ingrediënten hebben.

# *ShowS*

Vaak kunnen we ook werk uitstellen totdat we alle ingrediënten hebben. De prelude bevat het type:

**type** $\mathit{ShowS} = \mathit{String} \rightarrow \mathit{String}$

Het idee is dat we *String*s representeren m.b.v. functies die die string vooraan een andere *String* hangen, waarbij "concatenatie" gerepresenteerd wordt m.b.v. functiecompositie:

$r = ((\texttt{"aap"} +\!\!+) \circ (\texttt{"noot"} +\!\!+)) \circ (\texttt{"mies"} +\!\!+)$

# *ShowS*

Vaak kunnen we ook werk uitstellen totdat we alle ingrediënten hebben. De prelude bevat het type:

$$\textbf{type}\ \mathit{ShowS} = \mathit{String} \rightarrow \mathit{String}$$

Het idee is dat we *String*s representeren m.b.v. functies die die string vooraan een andere *String* hangen, waarbij "concatenatie" gerepresenteerd wordt m.b.v. functiecompositie:

$$r = ((\texttt{"aap"} +\!\!+) \circ (\texttt{"noot"} +\!\!+)) \circ (\texttt{"mies"} +\!\!+)$$

De eigenlijke *string*-waarde kijrgen we nu tevoorschijn door deze waarde $r$ op de lege string toe te passen:

$$r\ \texttt{""}$$

# *ShowS*

Vaak kunnen we ook werk uitstellen totdat we alle ingrediënten hebben. De prelude bevat het type:

**type** $ShowS = String \rightarrow String$

Het idee is dat we *String*s representeren m.b.v. functies die die string vooraan een andere *String* hangen, waarbij "concatenatie" gerepresenteerd wordt m.b.v. functiecompositie:

$$r = ((\texttt{"aap"} +\!\!+) \circ (\texttt{"noot"} +\!\!+)) \circ (\texttt{"mies"} +\!\!+)$$

De eigenlijke *string*-waarde kijrgen we nu tevoorschijn door deze waarde $r$ op de lege string toe te passen:

$$r\ \texttt{""}$$

De associativiteit van $\circ$ zorgt er nu voor dat het resultaat opgebouwd wordt in lineaire tijd.


© Universiteit Utrecht

# Halveren i.p.v. verminderen

Bekijk functies die zichzelf één keer aanroepen met constant vervolgwerk. Een voorbeeld daarvan is de machtsverhef-functie:

$$x \uparrow 0 = 1$$
$$x \uparrow n = x * x \uparrow (n-1)$$

Deze functie heeft complexiteit $\mathcal{O}(n)$.

# Halveren i.p.v. verminderen

Bekijk functies die zichzelf één keer aanroepen met constant vervolgwerk. Een voorbeeld daarvan is de machtsverhef-functie:

$$
\begin{aligned}
& x \uparrow 0 = 1 \\
& x \uparrow n = x * x \uparrow (n - 1)
\end{aligned}
$$

Deze functie heeft complexiteit $\mathcal{O}(n)$. Zou de parameter gehalveerd worden bij de recursieve aanroep, dan bedroeg de complexiteit slechts $\mathcal{O}(\log n)$. In het geval van machtsverheffen is dat mogelijk:

$$
\begin{aligned}
& x \uparrow 0 && = 1 \\
& x \uparrow n \mid \mathit{even}\ n && = \quad \mathit{kwadraat}\ (x \uparrow (n\ `\mathit{div}`\ 2)) \\
& x \uparrow n \mid \mathit{odd}\ n && = x * \mathit{kwadraat}\ (x \uparrow (n\ `\mathit{div}`\ 2)) \\
& \mathit{kwadraat}\ x && = x * x
\end{aligned}
$$

# De *ShowS*-truc nog een keer, maar dan verstopt ...

De functie *labels*, die de elementen van een boom in een lijst zet, heeft als de boom gebalanceerd is complexiteit $\mathcal{O}(n \log n)$:

$$
\begin{aligned}
&\mathit{labels} && :: \mathit{Boom}\ a \rightarrow [a] \\
&\mathit{labels}\ \mathit{Blad} && = [\,] \\
&\mathit{labels}\ (\mathit{Tak}\ x\ \mathit{li}\ \mathit{re}) && = \mathit{labels}\ \mathit{li} \mathbin{+\!\!+} [x] \mathbin{+\!\!+} \mathit{labels}\ \mathit{re}
\end{aligned}
$$

# De *ShowS*-truc nog een keer, maar dan verstopt ...

De functie *labels*, die de elementen van een boom in een lijst zet, heeft als de boom gebalanceerd is complexiteit $\mathcal{O}(n \log n)$:

$$
\begin{aligned}
&\mathit{labels} && :: \mathit{Boom}\ a \rightarrow [a] \\
&\mathit{labels}\ \mathit{Blad} && = [\,] \\
&\mathit{labels}\ (\mathit{Tak}\ x\ \mathit{li}\ \mathit{re}) && = \mathit{labels}\ \mathit{li} \mathbin{+\!\!+} [x] \mathbin{+\!\!+} \mathit{labels}\ \mathit{re}
\end{aligned}
$$

Ook nu is verbetering mogelijk, door een extra functie te maken, die eigenlijk te veel doet, en als volgt is gespecificeerd:

$$
\begin{aligned}
&\mathit{labelsVoor} && :: \mathit{Boom}\ a \rightarrow ([a] \rightarrow [a]) \\
&\mathit{labelsVoor}\ \mathit{boom}\ \mathit{xs} && = \mathit{labels}\ \mathit{boom} \mathbin{+\!\!+} \mathit{xs}
\end{aligned}
$$

Gegeven deze functie kan *labels* geschreven worden als:

$$labels\ boom = labelsVoor\ boom\ [\,]$$

Gegeven deze functie kan *labels* geschreven worden als:

$$labels\ boom = labelsVoor\ boom\ [\,]$$

Maar er is een andere definitie van *labelsVoor* mogelijk:

$$\begin{aligned} &labelsVoor\ Blad\ xs &&= xs \\ &labelsVoor\ (Tak\ x\ li\ re)\ xs &&= labelsVoor\ li\ (x : labelsVoor\ re\ xs) \end{aligned}$$

Het opbouwen van het resultaat gebeurt nu weer in lineaire tijd.

# Geheugengebruik

Behalve op de benodigde tijd kan een algoritme ook beoordeeld worden op de benodigde geheugenruimte.

# Geheugengebruik

Behalve op de benodigde tijd kan een algoritme ook beoordeeld worden op de benodigde geheugenruimte.

- het opslaan van het programma;

# Geheugengebruik

Behalve op de benodigde tijd kan een algoritme ook beoordeeld worden op de benodigde geheugenruimte.

- het opslaan van het programma;
- het opbouwen van datastructuren met behulp van constructorfuncties;

# Geheugengebruik

Behalve op de benodigde tijd kan een algoritme ook beoordeeld worden op de benodigde geheugenruimte.

- het opslaan van het programma;
- het opbouwen van datastructuren met behulp van constructorfuncties;
- het bijhouden van nog uit te rekenen deel-expressies.

# Geheugengebruik

Behalve op de benodigde tijd kan een algoritme ook beoordeeld worden op de benodigde geheugenruimte.

- het opslaan van het programma;
- het opbouwen van datastructuren met behulp van constructorfuncties;
- het bijhouden van nog uit te rekenen deel-expressies.

# Geheugengebruik

Behalve op de benodigde tijd kan een algoritme ook beoordeeld worden op de benodigde geheugenruimte.

- het opslaan van het programma;
- het opbouwen van datastructuren met behulp van constructorfuncties;
- het bijhouden van nog uit te rekenen deel-expressies.

Het geheugen voor de datastructuren wordt de heap ('hoop') genoemd; het geheugen voor de administratie van de berekening de stack ('stapel').

# Tijd en ruimtegebruik opvragen

Als er tijdens het uitrekenen van een expressie garbage collection(s) hebben plaatsgevonden, wordt dit vermeld (Hugs geeft ook het aantal garbage collections):

```
? length (subs [1..8])
256
(4141 reductions, 9286 cells, 1 garbage collection)
? length (subs [1..10])
1024
(18487 reductions, 43093 cells, 6 garbage collections)
?
```

# Tussenexpressies nemen ruimte

De stackruimte die nodig is om de tussenresultaten op te slaan kan groter zijn dan de uit te rekenen expressie en het resultaat. Bekijk bijvoorbeeld de tussenresultaten in de berekening $\mathit{foldr}\ (+)\ 0\ [1\,..\,5]$

$$
\begin{aligned}
&\mathit{foldr}\ (+)\ 0\ [1\,..\,5]\\
&1 + \mathit{foldr}\ (+)\ 0\ [2\,..\,5]\\
&1 + (2 + \mathit{foldr}\ (+)\ 0\ [3\,..\,5])\\
&1 + (2 + (3 + \mathit{foldr}\ (+)\ 0\ [4\,..\,5]))\\
&1 + (2 + (3 + (4 + \mathit{foldr}\ (+)\ 0\ [5])))\\
&1 + (2 + (3 + (4 + (5 + \mathit{foldr}\ (+)\ 0\ [\,]))))\\
&1 + (2 + (3 + (4 + (5 + 0))))\\
&1 + (2 + (3 + (4 + 5)))\\
&1 + (2 + (3 + 9))\\
&1 + (2 + 12)\\
&1 + 14\\
&15
\end{aligned}
$$

# We proberen het eens met *foldl*

Als je in plaats van *foldr* de functie *foldl* gebruikt worden de getallen op een andere manier opgeteld.

$$
\begin{array}{lll}
\mathit{foldl}\ (+) & 0 & [1\,..\,5] \\
\mathit{foldl}\ (+)\ ( & 0+1) & [2\,..\,5] \\
\mathit{foldl}\ (+)\ (( & 0+1)+2) & [3\,..\,5] \\
\mathit{foldl}\ (+)\ ((( & 0+1)+2)+3) & [4\,..\,5] \\
\mathit{foldl}\ (+)\ (((( & 0+1)+2)+3)+4) & [5] \\
\mathit{foldl}\ (+)\ ((((( & (0+1)+2)+3)+4)+5) & [\,] \\
 & (((( 0+1)+2)+3)+4)+5 & \\
 & (((\quad 1 \quad +2)+3)+4)+5 & \\
 & ((\quad 3 \quad +3)+4)+5 & \\
 & (\quad 6 \quad +4)+5 & \\
 & 10+5 & \\
 & 15 &
\end{array}
$$

# Zonder lazy evaluation

Zonder lazy evaluatie was de berekening als volgt verlopen; zodra dat mogelijk is wordt de tweede parameter van $foldl'$ uitgerekend:

$$
\begin{array}{ll}
foldl'\ (+)\ 0 & [1\,..\,5] \\
foldl'\ (+)\ (0+1) & [2\,..\,5] \\
foldl'\ (+)\ 1 & [2\,..\,5] \\
foldl'\ (+)\ (1+2) & [3\,..\,5] \\
foldl'\ (+)\ 3 & [3\,..\,5] \\
foldl'\ (+)\ (3+3) & [4\,..\,5] \\
foldl'\ (+)\ 6 & [4\,..\,5] \\
foldl'\ (+)\ (6+4) & [5] \\
foldl'\ (+)\ 10 & [5] \\
foldl'\ (+)\ (10+5) & [\,] \\
foldl'\ (+)\ 15 & [\,] \\
15 &
\end{array}
$$

# *seq*

Speciaal voor dit doel definieren we een functie , *strict*, gebruik makend van een ingebouwde functie $seq :: a \rightarrow b \rightarrow b$.

# *seq*

Speciaal voor dit doel definieren we een functie , *strict*, gebruik makend van een ingebouwde functie $seq :: a \to b \to b$. Wat *seq* doet is eerst zijn linker argument uitrekenen en vervolgens zijn rechter argument opleveren.

$$strict\ f\ x = x\ `seq`\ f\ x$$

De functie krijgt dus een functie en een waarde als parameter, en past de functie toe op de waarde, maar dwingt eerste wel even evaluatie van die waarde af.

# *foldl′*

In de prelude wordt een functie *foldl′* gedefinieerd, die hetzelfde doet als *foldl*, maar dan met niet-lazy tweede parameter.

# $foldl'$

In de prelude wordt een functie $foldl'$ gedefinieerd, die hetzelfde doet als $foldl$, maar dan met niet-lazy tweede parameter.

In de functie $foldl'$ wordt bij de recursieve aanroep de functie $strict$ gebruikt zoals hiervoor beschreven:

$$
\begin{aligned}
foldl'\ f\ e\ [\,] &= e \\
foldl'\ f\ e\ (x : xs) &= strict\ (foldl'\ f)\ (f\ e\ x)\ xs
\end{aligned}
$$

# *foldl′*

In de prelude wordt een functie *foldl′* gedefinieerd, die hetzelfde doet als *foldl*, maar dan met niet-lazy tweede parameter.

In de functie *foldl′* wordt bij de recursieve aanroep de functie *strict* gebruikt zoals hiervoor beschreven:

$$\begin{aligned} \mathit{foldl'}\ f\ e\ [\,] &= e \\ \mathit{foldl'}\ f\ e\ (x : xs) &= \mathit{strict}\ (\mathit{foldl'}\ f)\ (f\ e\ x)\ xs \end{aligned}$$

Gebruikmakend van *foldl′* is de stackruimte constant, zodat rustig de som van heel lange lijsten berekend kan worden:

```
? foldl (+) 0 [1..6000]
ERROR: Control stack overflow
? foldl' (+) 0 [1..6000]
18003000
```

# Meer prelude functies

In de prelude is dan ook gedefinieerd:

$$\begin{aligned} sum &= foldl'\ (+)\ 0 \\ product &= foldl'\ (*)\ 1 \end{aligned}$$

# Meer prelude functies

In de prelude is dan ook gedefinieerd:

$$\begin{aligned} sum &= foldl'\ (+)\ 0 \\ product &= foldl'\ (*)\ 1 \end{aligned}$$

Voor de operator $\wedge$ wordt echter wel *foldr* gebruikt:

$$and = foldr\ (\wedge)\ True$$

Bij deze operator zorgt lazy evaluatie er immers voor dat de tweede parameter niet wordt uitgerekend als dat niet nodig is.

# Haskell wetten

Voor allerlei functies is het mogelijk om wetten te formuleren waar ze aan voldoen. Zo geldt bijvoorbeeld voor alle functies $f$ en alle lijsten $xs$:

$$(map\ f \circ map\ g)\ xs = map\ (f \circ g)\ xs$$

Dit is geen *definitie* van $map$ of $\circ$ (die zijn al eerder gedefinieerd); het is een *wet* waar deze functies aan blijken te voldoen.

# Een paar belangrijke wetten

Een aantal belangrijke wetten die gelden voor de Haskell-standaardfuncties zijn de volgende:

- functiecompositie is associatief, dus

$$f \circ (g \circ h) = (f \circ g) \circ h$$

# Een paar belangrijke wetten

Een aantal belangrijke wetten die gelden voor de Haskell-standaardfuncties zijn de volgende:

- functiecompositie is associatief, dus

  $$f \circ (g \circ h) = (f \circ g) \circ h$$

- $map\ f$ distribueert over $+\!\!+$, dus

  $$map\ f\ (xs +\!\!+ ys) = map\ f\ xs +\!\!+ map\ f\ ys$$

# Een paar belangrijke wetten

Een aantal belangrijke wetten die gelden voor de Haskell-standaardfuncties zijn de volgende:

- functiecompositie is associatief, dus

  $$f \circ (g \circ h) = (f \circ g) \circ h$$

- $map\ f$ distribueert over $+\!\!+$, dus

  $$map\ f\ (xs +\!\!+ ys) = map\ f\ xs +\!\!+ map\ f\ ys$$

- de generalisatie hiervan naar een *lijst van* lijsten in plaats van *twee* lijsten:

  $$map\ f \circ concat = concat \circ map\ (map\ f)$$

# ... continued ...

- *map* distribueert over samenstelling:

$$map\ (f \circ g) = map\ f \circ map\ g$$

# ... continued ...

- *map* distribueert over samenstelling:

  $$map\ (f \circ g) = map\ f \circ map\ g$$

- Als $f$ associatief is (dus $x\ `f`\ (y\ `f`\ z) = (x\ `f`\ y)\ `f`\ z$, en $e$ is het neutrale element van $f$ (dus $x\ `f`\ e = e\ `f`\ x = x$ voor alle $x$), dan geldt voor eindige lijsten $xs$:

  $$foldr\ f\ e\ xs = foldl\ f\ e\ xs$$

# ... continued ...

- *map* distribueert over samenstelling:

  $$map\ (f \circ g) = map\ f \circ map\ g$$

- Als $f$ associatief is (dus $x\ `f`\ (y\ `f`\ z) = (x\ `f`\ y)\ `f`\ z$, en $e$ is het neutrale element van $f$ (dus $x\ `f`\ e = e\ `f`\ x = x$ voor alle $x$), dan geldt voor eindige lijsten $xs$:

  $$foldr\ f\ e\ xs = foldl\ f\ e\ xs$$

- Als bij *foldr* de beginwaarde het neutrale element van de operator is, dan is *foldr* over een singleton-lijst de identiteit:

  $$foldr\ f\ e\ [x] = x$$

# ... continued ...

- *map* distribueert over samenstelling:

  $$map\ (f \circ g) = map\ f \circ map\ g$$

- Als $f$ associatief is (dus $x\ `f`\ (y\ `f`\ z) = (x\ `f`\ y)\ `f`\ z$, en $e$ is het neutrale element van $f$ (dus $x\ `f`\ e = e\ `f`\ x = x$ voor alle $x$), dan geldt voor eindige lijsten $xs$:

  $$foldr\ f\ e\ xs = foldl\ f\ e\ xs$$

- Als bij *foldr* de beginwaarde het neutrale element van de operator is, dan is *foldr* over een singleton-lijst de identiteit:

  $$foldr\ f\ e\ [x] = x$$

- Een element op kop zetten van een lijst kan verwisseld worden met *map*, mits je dan de functiewaarde van het element op kop zet:

  $$map\ f \circ (x{:}) = (f\ x{:}) \circ map\ f$$

# ... continued ...

- Elk gebruik van *map* kan ook geschreven worden als aanroep van *foldr*:

$$map\ f\ xs = foldr\ g\ [\,]\ xs\ \textbf{where}\ g\ x\ ys = f\ x : ys$$

# Wetten bewijzen

**Wet** foldr over een singleton-lijst
*Als $e$ het neutrale element is van de operator $f$, dan geldt*

$$\mathit{foldr}\ f\ e\ [x] = x$$

# Wetten bewijzen

**Wet** foldr over een singleton-lijst
*Als $e$ het neutrale element is van de operator $f$, dan geldt*

$$foldr\ f\ e\ [x] = x$$

Bewijs:

```
foldr f e [x]
=   (notatie lijst-opsomming)
foldr f e (x:[])
=   (def. foldr)
f x (foldr f e [])
=   (def. foldr)
f x e
=   (voorwaarde van de wet)
x
```

# Nog een bwijs m.b.v equational reasoning

**Wet** functiecomposititie is associatief
*Voor alle functies $f$, $g$ en $h$ van het juiste type geldt:*

$$f \circ (g \circ h) = (f \circ g) \circ h$$

# Nog een bwijs m.b.v equational reasoning

**Wet** functiecompositite is associatief
*Voor alle functies $f$, $g$ en $h$ van het juiste type geldt:*

$$f \circ (g \circ h) = (f \circ g) \circ h$$

Bewijs:

```
(f . (g.h)) x
=   (def. (.))
f ((g.h) x)
=   (def. (.))
f (g (h x))
=   (def. (.))
(f.g) (h x)
=   (def. (.))
((f.g) . h) x
```

# Maar handiger is om in twee kolommen te werken

| | `f . ( g . h )` | `( f . g ) . h` |
|---|---|---|
| `x` | `(f . (g.h)) x`<br>= (def. (.))<br>`f ((g.h) x)`<br>= (def. (.))<br>`f (g (h x))` | `((f.g) . h) x`<br>= (def. (.))<br>`(f.g) (h x)`<br>= (def. (.))<br>`f (g (h x))` |

Van boven naar beneden werkend hebben geen fantasie nodig. De laatste regels zijn gelijk.

# Nog een voorbeeld

**Wet** map na op-kop

*Voor alle waarden $x$ en functies $f$ van het juiste type geldt:*

$$map\ f \circ (x{:}) = ((f\ x){:}) \circ map\ f$$

| | `map f . (x:)` | `((f x):) . map f` |
|---|---|---|
| `xs` | `(map f . (x:)) xs`<br>$=$ (def. `(.)`)<br>`map f ((x:) xs)`<br>$=$ (section-notatie)<br>`map f (x:xs)`<br>$=$ (def. `map`)<br>`f x : map f xs` | `( ((f x):) . map f) xs`<br>$=$ (def. `(.)`)<br>`((f x):) (map f xs)`<br>$=$ (section-notatie)<br>`f x : map f xs` |

# Inductieve bewijzen

Functies op lijsten hebben vaak een inductieve definitie. De functie wordt daarbij apart gedefinieerd voor $[\,]$. Dan wordt de functie gedefinieerd voor het patroon $(x : xs)$, waarbij de functie recursief aangeroepen mag worden op $xs$.

Bij het bewijs van wetten waarin eindige lijsten een rol spelen, kan ook inductie worden gebruikt. De wet wordt daarbij apart bewezen voor het geval dat de lijst $[\,]$ is. Vervolgens wordt de wet bewezen voor een lijst van de vorm $(x : xs)$. Bij dat bewijs mag al aangenomen worden dat de wet geldig is voor de lijst $xs$. Een voorbeeld van een wet die met inductie bewezen kan worden is de distributie van $map$ over $+\!\!+$.

# Bewijzen met inductie

**Wet** map na $+\!\!+$

$$map\ f\ (xs +\!\!+ ys) = map\ f\ xs +\!\!+ map\ f\ ys$$

Bewijs met inductie naar $xs$:

| xs | map f (xs++ys) | map f xs ++ map f ys |
|---|---|---|
| [] | map f ([]++ys)<br>= (def. ++)<br>map f ys | map f [] ++ map f ys<br>= (def. map)<br>[] ++ map f ys<br>= (def. ++)<br>map f ys |
| x:xs | map f ((x:xs)++ys)<br>= (def. ++)<br>map f (x:(xs++ys))<br>= (def. map)<br>f x : map f (xs++ys) | map f (x:xs) ++ map f ys<br>= (def. map)<br>(f x : map f xs) ++ map f ys<br>= (def. ++)<br>f x : (map f xs ++ map f ys) |

**Wet** map na functiecompositie

$$map\ (f \circ g) = map\ f \circ map\ g$$

Bewijs met inductie naar $xs$:

| | `map (f.g)` | `map f . map g` |
|---|---|---|
| `xs` | `map (f.g) xs` | `(map f . map g) xs`<br>$=$ (def. `(.)`)<br>`map f (map g xs)` |
| `[]` | `map (f.g) []`<br>$=$ (def. `map`)<br>`[]` | `map f (map g [])`<br>$=$ (def. `map`)<br>`map f []`<br>$=$ (def. `map`)<br>`[]` |
| `x:xs` | `map (f.g) (x:xs)`<br>$=$ (def. `map`)<br>`(f.g) x : map (f.g) xs`<br>$=$ (def. `(.)`)<br>`f(g x) : map (f.g) xs` | `map f (map g (x:xs))`<br>$=$ (def. `map`)<br>`map f (g x : map g xs)`<br>$=$ (def. `map`)<br>`f(g x) : map f (map g xs)` |