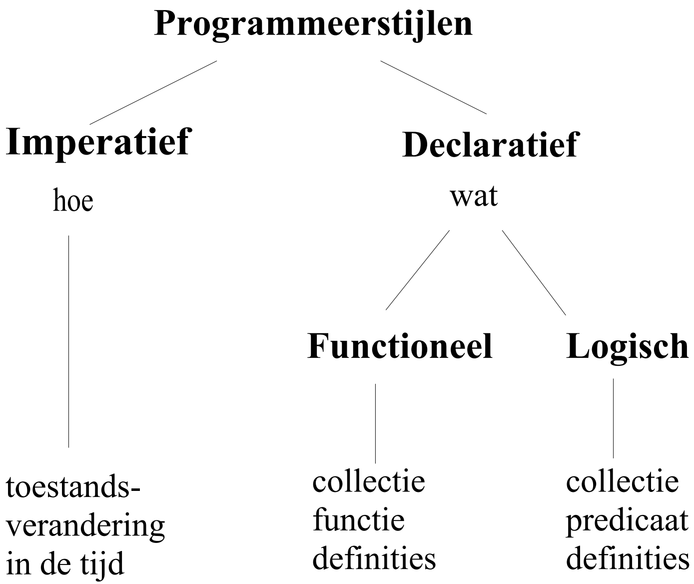
2IA05 Functioneel Programmeren

wk1: Introductie

Technische Universiteit **Eindhoven** University of Technology

Rik van Geldrop, Jaap van der Woude

Declarative programmeerstijl



FP-wetenswaardigheden

- Berust op wiskundige theorie (λ-calculus, herschrijfsysteem).
- Functies als argument/resultaat mogelijk (first class citizens).
- Functies met één argument: Currying.
- “Type en Structuur” zijn de leidraad bij ontwerp en analyse.
- Redeneren over programma’s/functies d.m.v. equational reasoning:
  - constructie/verificatie/transformatie/executie.
- Recursie i.p.v. repetitie.
- Compacte programma’s/specificaties.
- (Haskell) Type-systeem ondersteunt Polymorfie/Overloading.
- (Haskell) Lazy evaluation.

Notatie en terminologie

- Bij voorkeur de gebruikelijke wiskundige notatie voor functies (met applicatie en compositie als elementaire operaties), maar voor implementatie doeleinden `Haskell` (herkenbaar aan "teletype font").

Applicatie-voorbeeld: een "puntsgewijze" definitie van kwadrateren

wiskunde $sqr.x = x * x$ of $sqr(x) = x^2$

$(twice.f).x = f.(f.x)$

haskell

```
sqr x = x * x
twice f x = f (f x)
```

Compositie-voorbeeld: een "puntvrije" definitie van 4^e-machtsverheffing

wiskunde $quad = sqr \circ sqr$

haskell `quad = sqr.sqr` ☹

Notatie en terminologie

- In dit vak gaan we rekenen met functies. Puntsgewijs, maar ook puntvrij.
- De belangrijkste operatie hierbij is functie-gelijkheid:
  Definitie [Extensionaliteit]
  Voor alle functies f en g met hetzelfde domein geldt
  $$f = g \quad \equiv \quad \langle \forall x :: f.x = g.x \rangle \qquad \square$$
- Een andere nuttige operatie heet λ-abstractie:
  Definitie[λ-abstractie]
  Als x een variabele is van type X and $\mathcal{E}$ een expressie van type Y, dan heet $(\lambda x \to \mathcal{E})$ een λ-abstractie. Verder geldt
  $$(\lambda x \to \mathcal{E}) \in X \to Y$$
  en
  $$(\lambda x \to \mathcal{E}).a = \mathcal{E}(x := a) \qquad \text{voor alle } a \in X \qquad \square$$
  I.h.b. $f = \lambda x \to f.x$ (simplificatie)

puntsgewijs <-> puntvrij

Gebruik extensionaliteit en λ-abstractie.

- Van puntvrij naar puntsgewijs: extensionaliteit:

  Voor alle x

$$
\begin{array}{ll}
 & quad.x \\
= & \\
 & (sqr \circ sqr).x \\
= & \quad \{\text{def compositie}\} \\
 & sqr.(sqr.x)
\end{array}
$$

- Van puntsgewijs naar puntvrij: λ-abstractie en/of extensionaliteit:

$$sqr = \lambda\, x \longrightarrow x * x$$

  en

puntsgewijs <-> puntvrij

$$
\begin{array}{lll}
 & (twice.f).x & = \quad f.(f.x) \\
= & \quad \{\text{ abstractie }\} & \\
 & twice.f & = \quad \lambda x \rightarrow f.(f.x) \\
= & \quad \{\text{ compositie }\} & \\
 & twice.f & = \quad \lambda x \rightarrow (f \circ f).x \\
= & \quad \{\text{ simplificatie }\} & \\
 & twice.f & = \quad f \circ f \\
= & \quad \{\text{ abstractie }\} & \\
 & twice & = \quad \lambda f \rightarrow f \circ f
\end{array}
$$

Notationele conventies

Reduceer haakjes door het gebruik van de volgende conventies

- Applicatie heeft de hoogste prioriteit, dus

$$f \circ g.x = f \circ (g.x)$$

- Applicatie associeert naar links, dus

$$f.g.h = (f.g).h$$

- Abstractie associeert naar rechts, dus

$$\lambda x \rightarrow \lambda y \rightarrow E \;=\; \lambda x \rightarrow (\lambda y \rightarrow E)$$

Technische Universiteit Eindhoven University of Technology

Secties: speciale λ-abstracties

Zij $\oplus \in X \times Y \to Z$, $x \in X$ en $y \in Y$, dan

$$x \oplus y \in Z$$

Door λ-abstractie, bijvoorbeeld over y, ontstaat

$$\lambda y \to x \oplus y \quad \in \quad Y \to Z \qquad\qquad \text{bij vaste } x$$

Bij applicaties van deze functie verandert alleen het 2^e argument, i.e

$$(\lambda y \to x \oplus y).a = x \oplus a$$

Gebruikelijke afkorting is $(x\oplus)$ “sectie van $\oplus$”

- Idem voor $(\oplus y)$

Curry en uncurry

Currying is een functietransformatie die de isomorfie

$$X \times Y \to Z \quad \cong \quad X \to (Y \to Z)$$

aangeeft.

De constructie

1. Bepaal (uit de typering !) een functie, genaamd $curry$

$$curry : (X \times Y \to Z) \to (X \to (Y \to Z))$$

2. Bepaal (uit de typering !) een functie, genaamd $uncurry$

$$uncurry : (X \to (Y \to Z)) \to (X \times Y \to Z)$$

3. Ga na dat $curry \circ uncurry = id$ en $uncurry \circ curry = id$

Curry en uncurry

- Vanwege de isomorfie is het onbelangrijk welke “view” op functietypering gekozen wordt.
- In de wiskunde is een “ongecurriede” versie gebruikelijk. Bijvoorbeeld

  $$+ \quad \in \quad \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$

  en $+.(2,3)$

- Functionele talen prefereren een “gecurriede” versie omdat daarbij slechts een argument nodig is om voortgang van evaluatie/programmaexecutie te bewerkstelligen. In Haskell:

  ```
  (+) :: Int -> Int -> Int
  ```

  en `(+) 2 3`

  met verband `(+)` = $curry.+$

Type, polymorfie, overloading

basis • 1 unit type (singleton set)

samengesteld: Voor typen A en B

- $A \rightarrow B$ functieruimte
- $A \times B$ cartesisch product
  met projecties π_1 en π_2
- $A + B$ disjuncte som
  met injecties/constructoren in_1 en in_2

Polymorfie

Een polymorfe functie is één functie die op verschillende typen kan worden toegepast.Voorbeeld: $swap.(x, y) = (y, x)$

Overloading

Er is sprake van overloading als er aan verschillende functies dezelfde naam gegeven wordt. De keuze van de juiste definitie wordt gemaakt op grond van beschikbare environment-informatie. Voorbeeld: “=”

Haskell classes

Class: collectie typen die bepaalde overloaded operatoren ondersteunen.

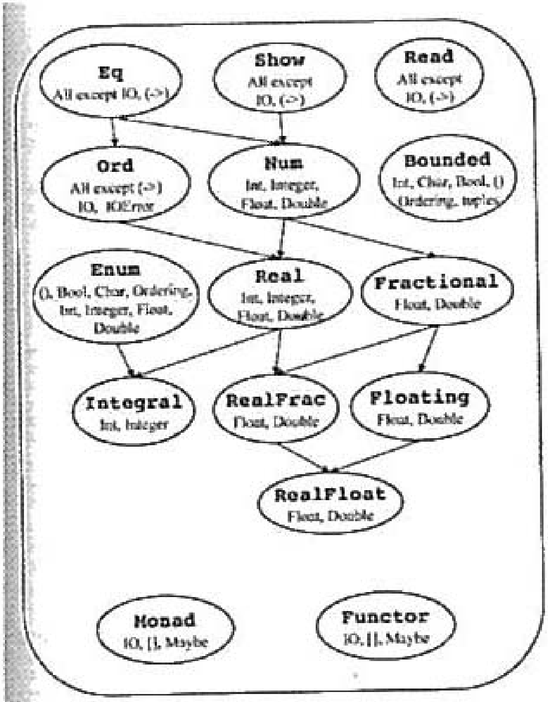


Figure 6.1: Standard Haskell Classes

Eq	(==)	::	$a \to a \to Bool$
	(/=)	::	$a \to a \to Bool$
Ord	(<)	::	$a \to a \to Bool$
	(<=)	::	$a \to a \to Bool$
	(>)	::	$a \to a \to Bool$
	(>=)	::	$a \to a \to Bool$
	min	::	$a \to a \to a$
	max	::	$a \to a \to a$
Show	*show*	::	$a \to String$
Read	*read*	::	$String \to a$
Num	(+)	::	$a \to a \to a$
	(−)	::	$a \to a \to a$
	(*)	::	$a \to a \to a$
	negate	::	$a \to a$
	abs	::	$a \to a$
	signum	::	$a \to a$
Integral	*div*	::	$a \to a \to a$
	mod	::	$a \to a \to a$
Fractional	(/)	::	$a \to a \to a$
	recip	::	$a \to a$
Bounded	*minBound*	::	a
	maxBound	::	a

Classes en instances

- class is te vergelijken met begrip “abstract class” in OO.
  Voorbeeld

```
class Eq where
  (==), (/=) :: a -> a -> Bool
  x /= y   =  not (x == y)
```

- Een type opnemen in een class door “completion” van operator-set.
  Bijvoorbeeld: type Void (i.e. `()`) opnemen in `Eq` door de ontbrekende definitie van `(==)` te geven

```
instance Eq () where
   () == () = True
```

Let bij class en instance definities op de offside-rule!!

Classes en instances, vervolg 1

- “extends”-relaties tussen classes.
  Voorbeeld: class `Ord` extends class `Eq`. Haskell weergave via “context”

```
class (Eq a) => Ord a where
  (<),(<=),(>),(>=) :: a -> a -> Bool
  min,max :: a -> a -> a
  -- definitie van (<) ontbreekt
  -- defs voor overige operatoren uitgedrukt in (<)
```

  - `(Eq a) =>` heet de “context” van `Ord`
  - Een type opnemen in `Ord` door
    - completion van `Eq`, i.e. definitie voor `(==)`
    - completion van `Ord`: definitie voor `(<)`

Classes en instances, vervolg 2

- “Multiple extension mogelijk.
  Bijvoorbeeld: class `Real` extends class `Ord` en extends class `Num`

```
class (Num a, Ord a) => Real a where
 ..............
```

- Instance definities mogelijk via het systeem (zie Sudoku.hs)
  - (via system)

```
data Stack a =  ......... deriving (Eq,Show)
```

  - (user-defined)

```
instance (Show a) => Show (Stack a) where
  show EmptyStk   = "-"
  show (Stk x s) = show x ++ " | " ++ show s
```


Technische Universiteit Eindhoven University of Technology

Overige Haskell taal elementen

geïllustreerd a.d.h.v. Sudoku programma

Technische Universiteit
Eindhoven
University of Technology

Statements

• GEEN Assignment		GEEN side-effect
• Sequentie	$\longmapsto$	fc-compositie, regel 126
• Alternatief statement		
if .. then .. else ..	$\longmapsto$	regel 75
case .. of ..	$\longmapsto$	regel 118
“guards”	$\longmapsto$	regel 83
• Repetitie	$\longmapsto$	recursie: - constructief, regel 119 - staartrecursief, regel 122
• Commentaar		– (tot eind regel), regel 5 {- meerdere regels -}
!! Offside-rule		“waar eindigt definitie”, regel 113

Definities

- Data/typen (standaard o.a. Int, Integer, Bool, Float)
  standaardtypen: `Int`, `Integer`, `Bool`, `Float`,..

`type`	type-synoniem	regel 12
`data`	nieuw type + constructoren	Stack.hs, regel 77
(`new type`)		

- GEEN subtypering — indien nodig: karakteriserend predikaat $\mathcal{P}(X) \cong (X \to \mathbb{B})$
- WEL (bijv) ['a' ..'z'] — consecutief deel van een geordende verzameling — regel 20
- Operator/Functie-definities:
  - puntvrij — regel 38
  - puntsgewijs — regel 44
  - \+ prioriteitsniveau
  - \+ associatie wijze

en Verder..

- Afkortingen

Lijstcomprehensie		regel 28
Pattern-matchen		regel 33
Lokale bindingen	let	regel 113
	where	regel 66

- Encapsulation

Module	hiding	regel..
	import	regel 3

- Compacte code — probleem: begrijpend lezen
- Uitgebreide Prelude

Technische Universiteit Eindhoven University of Technology