Universiteit Utrecht

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

College 2010-2011
7. Klassen en hun Instanties

Doaitse Swierstra

Utrecht University

October 5, 2010

Overloading versus Polymorfie

Veel functies en operatoren kunnen op waarden van verschillend type worden toegepast.

Overloading versus Polymorfie

Veel functies en operatoren kunnen op waarden van verschillend type worden toegepast. Er zijn twee mechanismen waardoor dat mogelijk is:

- Een polymorfe functie werkt op een bepaalde datastructuur (bijvoorbeeld lijsten), zonder gebruik te maken van eigenschappen van de elementen. De functie kan dus op datastructuren met een willekeurig element-type worden toegepast. Voorbeelden van polymorfe functies: *length*, *concat* en *map*.

Overloading versus Polymorfie

Veel functies en operatoren kunnen op waarden van verschillend type worden toegepast. Er zijn twee mechanismen waardoor dat mogelijk is:

- Een polymorfe functie werkt op een bepaalde datastructuur (bijvoorbeeld lijsten), zonder gebruik te maken van eigenschappen van de elementen. De functie kan dus op datastructuren met een willekeurig element-type worden toegepast. Voorbeelden van polymorfe functies: *length*, *concat* en *map*.
- Een overloaded functie kan op een aantal verschillende types werken die niets met elkaar te maken hebben. De operator $+$ werkt bijvoorbeeld zowel op type *Int* als op type *Float*. De operator $\leqslant$ kan op *Int* en *Float* werken, en daarnaast ook op *Char*, twee-tupels en lijsten.

Classes

Een klasse is een groep types waarop een bepaalde operator kan worden toegepast. De types Int en $Float$ vormen samen bijvoorbeeld Num, de klasse van numerieke types. De operator $(+)$ is op alle types in de klasse Num gedefinieerd.

Classes

Een klasse is een groep types waarop een bepaalde operator kan worden toegepast. De types Int en $Float$ vormen samen bijvoorbeeld Num, de klasse van numerieke types. De operator $(+)$ is op alle types in de klasse Num gedefinieerd. Dit komt tot uiting in het type van de operator $+$:

$$(+) :: Num\ a \Rightarrow a \rightarrow a \rightarrow a$$

Classes

Een klasse is een groep types waarop een bepaalde operator kan worden toegepast. De types Int en $Float$ vormen samen bijvoorbeeld Num, de klasse van numerieke types. De operator $(+)$ is op alle types in de klasse Num gedefinieerd. Dit komt tot uiting in het type van de operator $+$:

$$(+) :: Num\ a \Rightarrow a \rightarrow a \rightarrow a$$

De tekst $Num\ a$ kan gelezen worden als "type a zit in klasse Num". Het geheel heeft de betekenis: "$+$ heeft het type $a \rightarrow a \rightarrow a$ mits a in klasse Num zit".

... extra klassesn ...

Andere klassen die veel gebruikt worden zijn Eq en Ord. De klasse Eq is de klasse van types waarvan de elementen vergeleken kunnen worden; Ord is de klasse van ordenbare types. Operatoren die op types uit deze klassen gedefinieerd zijn, zijn bijvoorbeeld:

$$(\equiv) :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$$
$$(\leqslant) :: Ord\ a \Rightarrow a \rightarrow a \rightarrow Bool$$

Class en Instance declarations

Het is mogelijk om zelf nieuwe klassen te definiëren, naast de reeds bestaande klassen *Num*, *Eq* en *Ord*. Ook is het mogelijk om nieuwe types toe te voegen aan een klasse (zowel aan de drie bestaande klassen als aan zelf-gedefinieerde).

De klasse Num (eerste versie)

De definitie van de klasse *Num* is een goed voorbeeld van een klasse-declaratie. Deze ziet er, iets vereenvoudigd, als volgt uit:

```
class Num a where
  (+), (-), (*), (/) :: a→a→a
  negate             :: a→a
```

class declaraties

Een klasse-declaratie bestaat dus uit de volgende onderdelen:

- het (speciaal voor dit doel) gereserveerde woord **class**;
- de naam van de klasse (Num in het voorbeeld);
- een type-variabele (a in het voorbeeld);
- het gereserveerde woord **where**;
- type-declaraties voor operatoren en functies, waarbij de genoemde type-variabele gebruikt mag worden (en die op zich best weer polymorf mogen zijn!).

De instantie van Int voor Num

In een instance-declaratie wordt voor de aldus gedefinieerde operatoren een definitie gegeven.

De instantie van Int voor Num

In een instance-declaratie wordt voor de aldus gedefinieerde operatoren een definitie gegeven. De instance-declaratie waarmee wordt aangegeven dat *Int* in de klasse *Num* zit ziet er als volgt uit:

```
instance Num Int where
  (+)    = primPlusInt
  (-)    = primMinusInt
  (*)    = primMulInt
  (/)    = primDivInt
  negate = primNegInt
```

instance declaraties

Een instance-declaratie bestaat dus uit de volgende onderdelen:

- het gereserveerde woord **instance**;
- de naam van een klasse (*Num* in het voorbeeld);
- een type (*Int* in het voorbeeld);
- het gereserveerde woord **where**;
- definities voor de operatoren en functies die volgnes de klasse-declaratie aanwezig dienen te zijn.

De instantie van Float voor Num

```
instance Num Float where
  (+)    = primPlusFloat
  (-)    = primMinusFloat
  (*)    = primMulFloat
  (/)    = primDivFloat
  negate = primNegFloat
```

Nieuwe instanties

$$
\begin{aligned}
&\textbf{data } \mathit{Ratio} = \mathit{Rat}\ (\mathit{Int}, \mathit{Int}) \\
&\textbf{instance } \mathit{Num}\ \mathit{Ratio} \textbf{ where} \\
&\quad \mathit{Rat}\ (x,y) + \mathit{Rat}\ (p,q) = \mathit{eenvoud}\ (\mathit{Rat}\ (x*q + y*p, y*q)) \\
&\quad \mathit{Rat}\ (x,y) - \mathit{Rat}\ (p,q) = \mathit{eenvoud}\ (\mathit{Rat}\ (x*q - y*p, y*q)) \\
&\quad \mathit{Rat}\ (x,y) * \mathit{Rat}\ (p,q) \ = \mathit{eenvoud}\ (\mathit{Rat}\ (x*p, y*q)) \\
&\quad \mathit{Rat}\ (x,y)\ /\ \mathit{Rat}\ (p,q) \ = \mathit{eenvoud}\ (\mathit{Rat}\ (x*q, y*p)) \\
&\quad \mathit{negate}\ (\mathit{Rat}\ (x,y)) \quad\ = \mathit{Rat}\ (\mathit{negate}\ x, y)
\end{aligned}
$$

Nieuwe instanties

Ook de polynomen (die we vorige keer hebben gezien) kunnen een instantie van *Num* gemaakt worden:

```
data Poly  = Poly [Term]
data Term = Term (Float, Int)

instance Num Poly where
    (+) = pPlus
    (-) = pMin
    (*) = pMaal
    (/) = error "Geen deling gedefinieerd"
                -- gebruik dit spaarzaam
```

Constanten

Door het klasse-mechanisme kan voor optelling de operator + gebruikt worden, ongeacht of de op te tellen waardes integers zijn, *Float*'s, of zelfgedefinieerde numerieke types, zoals *Ratio* of *Complex*.

Constanten

Door het klasse-mechanisme kan voor optelling de operator + gebruikt worden, ongeacht of de op te tellen waardes integers zijn, *Float*'s, of zelfgedefinieerde numerieke types, zoals *Ratio* of *Complex*. Lastig is echter, dat het voor de notatie van constanten wèl belangrijk is wat het gewenste type is.

Constanten

Door het klasse-mechanisme kan voor optelling de operator $+$ gebruikt worden, ongeacht of de op te tellen waardes integers zijn, *Float*'s, of zelfgedefinieerde numerieke types, zoals *Ratio* of *Complex*. Lastig is echter, dat het voor de notatie van constanten wèl belangrijk is wat het gewenste type is. Zo moet voor de waarde 'drie' geschreven worden:

met het type Int:	3
met het type Float:	3.0
met het type Ratio:	$Rat\ (3, 1)$
met het type Complex:	$Comp\ (3.0, 0.0)$

Als bijvoorbeeld is gedefinieerd: $half = Rat\ (1,2)$, dan kun je niet schrijven:

$$3 * half$$

Als bijvoorbeeld is gedefinieerd: $half = Rat\ (1,2)$, dan kun je niet schrijven:

$$3 * half$$

De waarde *half* heeft immers het type *Ratio*, terwijl 3 het type *Int* heeft.

Het is bijvoorbeeld wel mogelijk om een overloaded functie *verdubbel* te schrijven, maar een functie *halveer* lukt niet.

Het is bijvoorbeeld wel mogelijk om een overloaded functie *verdubbel* te schrijven, maar een functie *halveer* lukt niet. Het enige wat er op zou zitten is om hier verschillende versies van te maken:

Het is bijvoorbeeld wel mogelijk om een overloaded functie *verdubbel* te schrijven, maar een functie *halveer* lukt niet. Het enige wat er op zou zitten is om hier verschillende versies van te maken:

$$
\begin{aligned}
&\mathit{verdubbel} &&:: \mathit{Num}\ a \Rightarrow a \rightarrow a\\
&\mathit{verdubbel} \quad x &&= x + x\\
&\mathit{halveerInt} &&:: \mathit{Int} \rightarrow \mathit{Int}\\
&\mathit{halveerFloat} &&:: \mathit{Float} \rightarrow \mathit{Float}\\
&\mathit{halveerInt} \quad n &&= n \mathbin{/} 2\\
&\mathit{halveerFloat}\ x &&= x \mathbin{/} 2.0
\end{aligned}
$$

De klasse Num opnieuw

Om het probleem beter op te lossen, is er in de prelude voor gekozen om nog één functie aan de klasse *Num* toe te voegen: de functie *fromInteger*. De volledige klasse-declaratie luidt dus:

```
class Num a where
  (+), (-), (*), (/) :: a→a→a
  negate             :: a→a
  fromInteger        :: Int→a
```

De klasse Num opnieuw

Om het probleem beter op te lossen, is er in de prelude voor gekozen om nog één functie aan de klasse *Num* toe te voegen: de functie *fromInteger*. De volledige klasse-declaratie luidt dus:

```
class Num a where
  (+), (-), (*), (/) :: a→a→a
  negate             :: a→a
  fromInteger        :: Int→a
```

Hiermee wordt gespecificeerd dat er voor elk instance-type van *Num* een conversie-functie moet zijn van *Int* naar dat type.

Voor zelf gedefinieerde types dus:

Voor zelf-gedefinieerde types is het nu nodig om zelf een definitie van *fromInteger* definiëren, bijvoorbeeld:

```
instance Num Ratio where
    ....
    fromInteger n =    Rat (n,1)
```

Eq

In de prelude wordt een klasse Eq gedefinieerd. De instances van deze klasse zijn de types waarvan de elementen met elkaar vergeleken kunnen worden. De klasse-declaratie is als volgt:

```
class Eq a where
  (≡), (≢) :: a→a→Bool
```

Verborgen gedefinieerd:

```
instance Eq Int where
    x ≡ y = primEqInt x y
    x ≢ y = ¬ (x ≡ y)
instance Eq Float where
    x ≡ y = primEqFloat x y
    x ≢ y = ¬ (x ≡ y)
instance Eq Char where
    x ≡ y = ord x ≡ ord y
    x ≢ y = ¬ (x ≡ y)
instance Eq Bool  where
    True ≡  True  = True
    _    ≡  _     = False
    x ≢ y         = ¬ (x ≡ y)
```

Eq opnieuw

class *Eq a* **where**

$$(\equiv), (\not\equiv) :: a \to a \to Bool$$
$$x \not\equiv y = \neg\ (x \equiv y)$$
$$x \equiv y = \neg\ (x \not\equiv y)$$

De definitie van $\not\equiv$ in de instance-declaraties kan nu worden weggelaten, omdat de default-definities de ontbrekende functies zelf invullen.

Ratio als instantie van Eq

Ook zelfgedefinieerde types kunnen tot instance van *Eq* gemaakt worden.

instance *Eq Ratio* **where**

$$Rat\ (x,y) \equiv Rat\ (p,q) = x * q \equiv y * p$$

Ratio als instantie van Eq

Ook zelfgedefinieerde types kunnen tot instance van *Eq* gemaakt worden.

instance *Eq Ratio* **where**

$$Rat\ (x,y) \equiv Rat\ (p,q) = x * q \equiv y * p$$

De gelijkheid die in de rechterkant van de definitie gebruikt wordt is de gelijkheid tussen integers. De vertaler kan dat afleiden aan de hand van de typering. Een definitie van $\not\equiv$ kan, net als in de instance-declaraties in de prelude, achterwege blijven. De default-definitie is ook in dit geval immers bruikbaar.

Superclasses

In de klasse-declaratie voor Ord wordt aangegeven dat een type ook een instance van Eq moet zijn, wil het ordenbaar zijn:

$$\begin{array}{l}\textbf{class } Eq\ a \Rightarrow Ord\ a \textbf{ where} \\ \quad (\leqslant), (<), (\geqslant), (>) :: a \to a \to Bool \\ \quad max, min \qquad\qquad\quad :: a \to a \to a\end{array}$$

Alle operatoren en functies in deze klasse met uitzondering van $\leqslant$ hebben een default-definitie.

Superclasses

In de klasse-declaratie voor Ord wordt aangegeven dat een type ook een instance van Eq moet zijn, wil het ordenbaar zijn:

$$\begin{array}{l}\textbf{class } Eq\ a \Rightarrow Ord\ a \textbf{ where} \\ \quad (\leqslant), (<), (\geqslant), (>) :: a \to a \to Bool \\ \quad max, min \qquad\qquad :: a \to a \to a\end{array}$$

Alle operatoren en functies in deze klasse met uitzondering van $\leqslant$ hebben een default-definitie. De enige operator die in instance-declaraties gedefinieerd hoeft te worden is dus $\leqslant$. Het is vanwege deze default-definities dat geëist wordt dat instances van Ord ook instances van Eq zijn.

De klasse Num

Net als *Ord* eist de klasse-declaratie voor *Num* in de prelude dat de instances ook een instance van de klasse *Eq* zijn. De, nu helemaal complete klasse-declaratie voor *Num* luidt derhalve:

```
class Eq a ⇒ Num a where
  (+), (−), (∗), (/) :: a→a→a
  negate             :: a→a
  fromInteger        :: Int→a
```

Geparametriseerde Instanties

Ook bij instance-declaraties kan als voorwaarde worden opgegeven, dat een bepaald type deel uitmaakt van een bepaalde klasse.

Geparametriseerde Instanties

Ook bij instance-declaraties kan als voorwaarde worden opgegeven, dat een bepaald type deel uitmaakt van een bepaalde klasse. Deze constructie wordt o.a. gebruikt om in één keer alle denkbare lijsten tot instance van Eq te maken:

instance $Eq\ a \Rightarrow Eq\ [a]$ **where**

$$
\begin{array}{lll}
[\,] & \equiv [\,] & = \mathit{True} \\
[\,] & \equiv (y : ys) & = \mathit{False} \\
(x : xs) & \equiv [\,] & = \mathit{False} \\
(x : xs) & \equiv (y : ys) & = x \equiv y \wedge xs \equiv ys
\end{array}
$$

Geparametriseerde Instanties

Ook bij instance-declaraties kan als voorwaarde worden opgegeven, dat een bepaald type deel uitmaakt van een constructie wordt o.a. gebruikt om in één ten tot instance van Eq te maken:

voor een enkel element

$$
\begin{array}{l}
\textbf{instance}\ Eq\ a \Rightarrow Eq\ [a]\ \textbf{where} \\
\quad [\,] \equiv [\,] = True \\
\quad [\,] \equiv (y : ys) = False \\
\quad (x : xs) \equiv [\,] = False \\
\quad (x : xs) \equiv (y : ys) = x \equiv y \wedge xs \equiv ys
\end{array}
$$

(ISA)

Geparametriseerde Instanties

Ook bij instance-declaraties kan als voorwaarde worden opgegeven, dat een bepaald type deel uitmaakt van een
constru
één
ten tot i

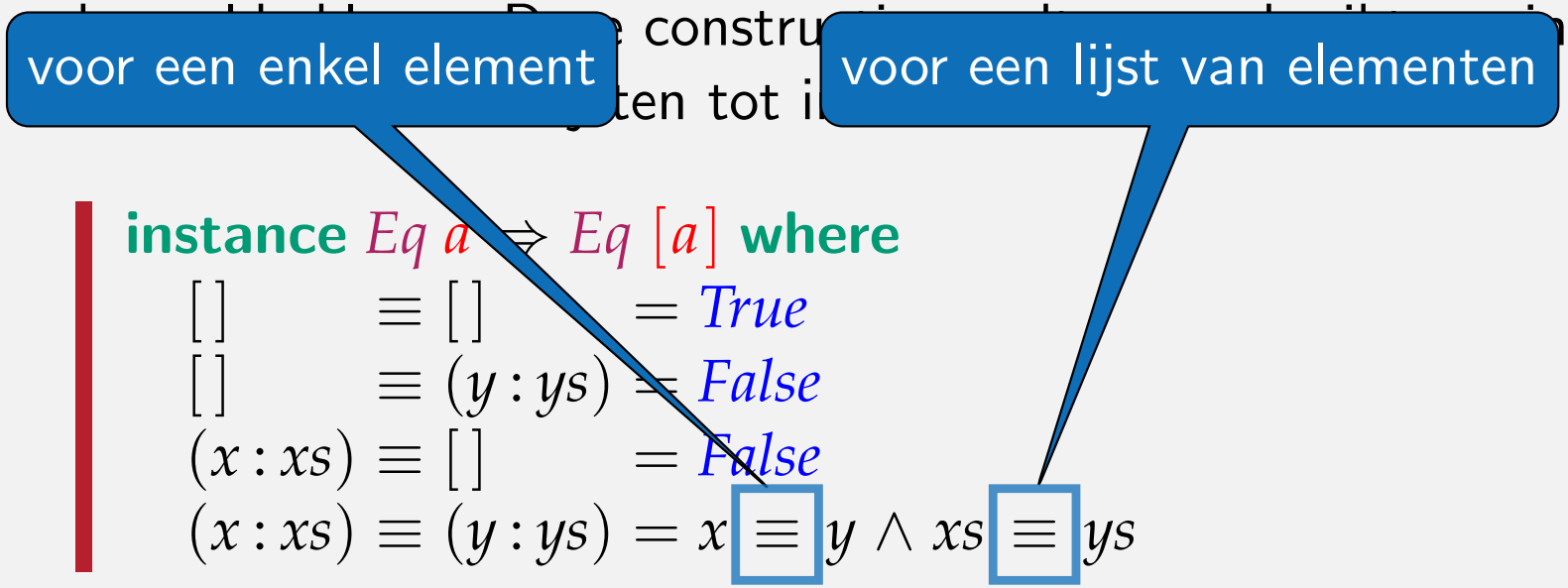
voor een enkel element

voor een lijst van elementen

$$
\begin{array}{l}
\textbf{instance}\ Eq\ a \Rightarrow Eq\ [a]\ \textbf{where} \\
\quad [\,] \equiv [\,] = True \\
\quad [\,] \equiv (y : ys) = False \\
\quad (x : xs) \equiv [\,] = False \\
\quad (x : xs) \equiv (y : ys) = x \equiv y \wedge xs \equiv ys
\end{array}
$$

(ISA) (ISLA)

Geparametriseerde Instanties

Ook bij instance-declaraties kan als voorwaarde worden opgegeven, dat een bepaald type deel uitmaakt van een


(ISA) (ISLA) De eerste regel van deze instance-declaratie kan zo gelezen worden: 'als a een instance is van Eq, dan is ook $[a]$ een instance van Eq'.

Nog een voorbeeld

Elders werd al opgemerkt dat lijsten een ordening kennen: de lexicografische ordening. Deze ordening wordt gedefinieerd door een instance-declaratie in de prelude:

$$
\begin{aligned}
&\textbf{instance } \mathit{Ord}\ a \Rightarrow \mathit{Ord}\ [a] \textbf{ where} \\
&\quad [\,] \leqslant ys \qquad\qquad = \mathit{True} \\
&\quad (x : xs) \leqslant [\,] \qquad = \mathit{False} \\
&\quad (x : xs) \leqslant (y : ys) = x < y \vee (x \equiv y \wedge xs \leqslant ys)
\end{aligned}
$$

Meerdere parameters/voorwaarden

Een instance-declaratie kan meer dan één voorwaarde hebben. Die moeten dan tussen haakjes genoteerd worden, met komma's ertussen.

Meerdere parameters/voorwaarden

Een instance-declaratie kan meer dan één voorwaarde hebben. Die moeten dan tussen haakjes genoteerd worden, met komma's ertussen.Hiermee kan bijvoorbeeld de gelijkheid van twee-tupels gedefinieerd worden, zoals in de prelude gebeurt:

instance $(Eq\ a, Eq\ b) \Rightarrow Eq\ (a,b)$ **where**
$(x,y) \equiv (u,v) = x \equiv u \wedge y \equiv v$

De elementen van een tweetupel (a,b) zijn dus vergelijkbaar mits zowel a als b een instance is van Eq, en dus vergelijkbaar zijn.

Let op!

Het pijltje met dubbele stok ($\Rightarrow$) kan gebruikt worden in type-declaraties, in instance-declaraties en in klasse-declaraties. Let op het verschil in betekenis van deze drie vormen:

- $f :: Num\ a \Rightarrow a \rightarrow a$ is een type-declaratie: f is een functie met type $a \rightarrow a$ mits a een type is in de klasse Num.

Let op!

Het pijltje met dubbele stok ($\Rightarrow$) kan gebruikt worden in type-declaraties, in instance-declaraties en in klasse-declaraties. Let op het verschil in betekenis van deze drie vormen:

- $f :: Num\ a \Rightarrow a \rightarrow a$ is een type-declaratie: f is een functie met type $a \rightarrow a$ mits a een type is in de klasse Num.
- **instance** $Eq\ a \Rightarrow Eq\ [a]$ is een instance-declaratie: $[a]$ is een instance van Eq mits a dat ook is.

Let op!

Het pijltje met dubbele stok ($\Rightarrow$) kan gebruikt worden in type-declaraties, in instance-declaraties en in klasse-declaraties. Let op het verschil in betekenis van deze drie vormen:

- $f :: Num\ a \Rightarrow a \rightarrow a$ is een type-declaratie: f is een functie met type $a \rightarrow a$ mits a een type is in de klasse Num.
- **instance** $Eq\ a \Rightarrow Eq\ [a]$ is een instance-declaratie: $[a]$ is een instance van Eq mits a dat ook is.
- **class** $Eq\ a \Rightarrow Ord\ a$ is een klasse-declaratie: alle instances van de nieuwe klasse Ord moeten ook instances zijn van Eq.

Standaard klassen

In de prelude worden de volgende klassen gedefinieerd:

- *Eq*, de klasse van vergelijkbare types;
- *Ord*, de klasse van ordenbare types;
- *Num*, de klasse van numerieke types;
- *Enum*, de klasse van opsombare types;
- *Ix*, de klasse van index-types;
- *Text*, de klasse van afdrukbare types.

Enum

De klasse *Enum* is als volgt gedefinieerd:

```
class Ord a ⇒ Enum a where
  enumFrom     :: a       → [a]
  enumFromThen :: a → a   → [a]
  enumFromTo   :: a → a   → [a]
  enumFromThenTo :: a → a → a → [a]
```

Gebruik van Enum

Een paar voorbeelden van het resultaat van de functies voor verschillende types:

$$
\begin{aligned}
&\mathit{enumFrom}\ 4 && = [4,5,6,7,8,\ldots \\
&\mathit{enumFromTo}\ \texttt{'c'}\ \texttt{'f'} && = [\texttt{'c'},\texttt{'d'},\texttt{'e'},\texttt{'f'}] \\
&\mathit{enumFromThenTo}\ 1.0\ 1.5\ 3.0 && = [1.0,1.5,2.0,2.5,3.0]
\end{aligned}
$$

Gebruik van Enum

Een paar voorbeelden van het resultaat van de functies voor verschillende types:

$$
\begin{aligned}
&\mathit{enumFrom}\ 4 && = [4,5,6,7,8,\ldots \\
&\mathit{enumFromTo}\ \texttt{'c'}\ \texttt{'f'} && = [\texttt{'c'},\texttt{'d'},\texttt{'e'},\texttt{'f'}] \\
&\mathit{enumFromThenTo}\ 1.0\ 1.5\ 3.0 && = [1.0,1.5,2.0,2.5,3.0]
\end{aligned}
$$

Deze vier functies worden gebruikt om de speciale notaties $[x..]$, $[x,y..]$, $[x..y]$ en $[x,y..z]$ een betekenis te geven.

De instance-declaratie *Enum Int* luidt als volgt:

instance *Enum Int* **where**

$$\begin{aligned} &\mathit{enumFrom}\ n = &&\mathit{iterate}\ (1+)\ n \\ &\mathit{enumFromThen}\ n\ m = &&\mathit{iterate}\ ((m-n)+)\ n \end{aligned}$$

Voor de functies *enumFromTo* en *enumFromThenTo* is er een default-definitie in de klasse-declaratie:

```
class Ord a ⇒ Enum a where
  ....
  enumFromTo       n m  = takeWhile (m ⩾) (enumFrom n)
  enumFromThenTo n n' m
      | n' > n          = takeWhile (m ⩾)
                                    (enumFromThen n n')
      | otherwise       = takeWhile (m ⩽)
                                    (enumFromThen n n')
```

Voor de functies *enumFromTo* en *enumFromThenTo* is er een default-definitie in de klasse-declaratie:

```
class Ord a ⇒ Enum a where
  ....
  enumFromTo       n m  = takeWhile (m ⩾) (enumFrom n)
  enumFromThenTo n n' m
      | n' > n          = takeWhile (m ⩾)
                                    (enumFromThen n n')
      | otherwise       = takeWhile (m ⩽)
                                    (enumFromThen n n')
```

Omdat in deze definities de ordenings-operatoren gebruikt worden, is het noodzakelijk dat elke instance van *Enum* ook een instance is van *Ord*.

Ix

De klasse *Ix* lijkt op *Enum*. De klasse-declaratie is als volgt:

```
class Ord a ⇒ Ix a where
  range   :: (a,a)      →[a]
  index   :: (a,a)→a→Int
  inRange :: (a,a)→a→Bool
```

Int is een instantie van Ix

instance *Ix Int* **where**

$$
\begin{aligned}
&\mathit{range} \quad (m,n) \quad = [m\,..\,n] \\
&\mathit{index} \quad (m,n)\ i = i - m \\
&\mathit{inRange}\ (m,n)\ i = m \leqslant i \wedge i \leqslant n
\end{aligned}
$$

Text

De klasse *Text* introduceert twee functies, waarmee respectievelijk een element en een lijst van het instance-type omgezet kunnen worden in een *String*.

Text

De klasse *Text* introduceert twee functies, waarmee respectievelijk een element en een lijst van het instance-type omgezet kunnen worden in een *String*. Deze functies worden zelden direct gebruikt. Meestal wordt in plaats daarvan de functie *show* gebruikt, die in de prelude wordt gedefinieerd:

$$show :: Text\ a \Rightarrow a \rightarrow String$$

Text

De klasse *Text* introduceert twee functies, waarmee respectievelijk een element en een lijst van het instance-type omgezet kunnen worden in een *String*. Deze functies worden zelden direct gebruikt. Meestal wordt in plaats daarvan de functie *show* gebruikt, die in de prelude wordt gedefinieerd:

$$show :: Text\ a \Rightarrow a \rightarrow String$$

Met de functie *show* kan elke waarde van een type dat een instance is van *Text* worden omgezet naar een *String*. Standaard-instances van *Text* zijn *Int*, *Float*, *Char* en lijsten en tweetupels waarvan de element-types ook instances zijn van *Text*.

Probleem 1: Cannot derive instance

Deze foutmelding is het gevolg als je een operator uit een klasse gebruikt met parameters waarvoor er geen instance is gedeclareerd. Bijvoorbeeld:

```
? (1,2,3) == (4,5,6) ERROR: Cannot derive
instance in expression *** Expression :
(1,2,3) == (4,5,6) *** Required instance :
Eq (Int,Int,Int)
```

Probleem 1: Cannot derive instance

Deze foutmelding is het gevolg als je een operator uit een klasse gebruikt met parameters waarvoor er geen instance is gedeclareerd. Bijvoorbeeld:

```
? (1,2,3) == (4,5,6) ERROR: Cannot derive
instance in expression *** Expression :
(1,2,3) == (4,5,6) *** Required instance :
Eq (Int,Int,Int)
```

In de prelude staat geen declaratie waarmee drietupels tot instance van *Eq* gemaakt worden (wel voor tweetupels en lijsten). Daarom kan de operator $\equiv$ niet zonder meer op drietupels toegepast worden.

Deze foutmelding wordt ook gegeven als je functies probeert te vergelijken. Functie-types zijn immers geen instance van *Eq*:

```
? tail == drop 1 ERROR: Cannot derive instance
in expression *** Expression : tail == drop 1
*** Required instance : Eq ([a]->[a])
```

Overlapping instances

Het is niet mogelijk om twee declaraties te geven waarmee een type een instantie wordt van dezelfde klasse. Bij gebruik van een operator op een waarde van dat type treedt er een ambiguiteit op!

Hetzelfde probleem treedt op als het type in een instance-declaratie een speciaal geval is van een type waarvoor al een andere instance-declaratie bestaat.

Hetzelfde probleem treedt op als het type in een instance-declaratie een speciaal geval is van een type waarvoor al een andere instance-declaratie bestaat. Bijvoorbeeld: in de prelude worden tweetupels gedeclareerd als instance van Eq:

instance $(Eq\ a, Eq\ b) \Rightarrow Eq\ (a,b)$ **where**

$$(x,y) \equiv (u,v) = x \equiv u \wedge y \equiv v$$

Hetzelfde probleem treedt op als het type in een instance-declaratie een speciaal geval is van een type waarvoor al een andere instance-declaratie bestaat. Bijvoorbeeld: in de prelude worden tweetupels gedeclareerd als instance van Eq:

```
instance (Eq a, Eq b) => Eq (a, b) where
  (x, y) == (u, v) = x == u && y == v
```

Als rationale getallen als tweetupel van twee integers worden gedefinieerd, zou je daarop wel een andere gelijkheid willen definiëren:

```
type Ratio = (Int, Int)
instance Eq Ratio  where
  (x, y) == (u, v) = x * v == u * y
```

Bij een aanroep van $(1,2) \equiv (2,4)$ kan nu niet gekozen worden: wordt de standaard tupel-gelijkheid bedoeld of de *Ratio*-gelijkheid?

Bij een aanroep van $(1,2) \equiv (2,4)$ kan nu niet gekozen worden: wordt de standaard tupel-gelijkheid bedoeld of de *Ratio*-gelijkheid? Bij het analyseren van de instance-declaratie wordt daarom een foutmelding gegeven:

```
ERROR "file" (line 12): Overlapping instances
for class "Eq" *** This instance : Eq (Int,Int)
*** Overlaps with : Eq (a,a) *** Common
instance : Eq (Int,Int)
```

Bij een aanroep van $(1,2) \equiv (2,4)$ kan nu niet gekozen worden: wordt de standaard tupel-gelijkheid bedoeld of de *Ratio*-gelijkheid? Bij het analyseren van de instance-declaratie wordt daarom een foutmelding gegeven:

```
ERROR "file" (line 12): Overlapping instances
for class "Eq" *** This instance : Eq (Int,Int)
*** Overlaps with : Eq (a,a) *** Common
instance : Eq (Int,Int)
```

Dit probleem kan worden opgelost door types waar een 'rare' gelijkheid op gedefinieerd moet worden als beschermd type te definiëren, dus met gebruikmaking van **data** in plaats van **type**:

data $Ratio = Rat\ (Int, Int)$

Probleem 3: Unresolved overloading

Bij het gebruik van een overloaded operator besluit de interpreter op grond van de types van de parameters welke definitie gekozen moet worden. Zo wordt in $1 + 2$ de integer-versie van $+$ gebruikt, en in $1.0 + 2.0$ de float-versie.

Probleem 3: Unresolved overloading

Bij het gebruik van een overloaded operator besluit de interpreter op grond van de types van de parameters welke definitie gekozen moet worden. Zo wordt in $1 + 2$ de integer-versie van $+$ gebruikt, en in $1.0 + 2.0$ de float-versie. Maar als de parameters zelf het resultaat zijn van een overloaded functie, kan het zijn dat er meerdere mogelijkheden zijn. Dat is bijvoorbeeld het geval in de volgende expressie:

```
? fromInteger 1 + fromInteger 2 ERROR:
Unresolved overloading *** type : Num a =>
a
```

Probleem 3: Unresolved overloading

Bij het gebruik van een overloaded operator besluit de interpreter op grond van de types van de parameters welke definitie gekozen moet worden. Zo wordt in $1 + 2$ de integer-versie van $+$ gebruikt, en in $1.0 + 2.0$ de float-versie. Maar als de parameters zelf het resultaat zijn van een overloaded functie, kan het zijn dat er meerdere mogelijkheden zijn. Dat is bijvoorbeeld het geval in de volgende expressie:

```
? fromInteger 1 + fromInteger 2 ERROR:
Unresolved overloading *** type : Num a =>
a
```

Voor *fromInteger* kan de integer-versie of de float-versie gekozen worden.

Meestal is deze fout te herstellen door de interpreter een extra hint te geven over het gewenste type. Dat kan bijvoorbeeld door een type-declaratie te geven voor de kritieke functies, of door de dubieuze expressie direct te typeren:

```
? fromInteger 1 + fromInteger 2 :: Int 3
```

Functor

De functie $map :: (a \to b) \to [a] \to [b]$ wordt in de prelude gegeneraliseerd:

```
class Functor f where
  fmap :: (a→b)→f a→f b
```

Hiermee wordt het patroon gevat dat elk type van de vorm $f\ a$ kennelijk ergens waarden van type a bevat, en die kunnen op een b afgebeeld worden.

Functor

De functie $map :: (a \to b) \to [a] \to [b]$ wordt in de prelude gegeneraliseerd:

```
class Functor f where
  fmap :: (a→b)→f a→f b
```

Hiermee wordt het patroon gevat dat elk type van de vorm $f\ a$ kennelijk ergens waarden van type a bevat, en die kunnen op een b afgebeeld worden.

```
instance Functor ((→) c) where
  fmap :: (a→b)→((→) c a)→((→) c b)
  fmap f c2a = f ∘ c2a
```

Functor

De functie $map :: (a \to b) \to [a] \to [b]$ wordt in de prelude gegeneraliseerd:

```
class Functor f where
  fmap :: (a→b)→f a→f b
```

Hiermee wordt het patroon gevat dat elk type van de vorm $f\ a$ kennelijk ergens waarden van type a bevat, en die kunnen op een b afgebeeld worden.

```
instance Functor ((→) c) where
  fmap :: (a→b)→((→) c a)→((→) c b)
  fmap f c2a = f ∘ c2a
```

Dus de $\circ$ operator is eigenlijk *map*, maar dan voor functies!

Monad

Een tweede belangrijke klasse die ten grondslag ligt aan veel instanties is de klasse:

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>)  :: m a -> m b        -> m b
  return :: b                -> m b
  fail   :: String           -> m b

  mf >> ma = fm >>= \_ -> ma
  fail s = error s
```

Monad

Een tweede belangrijke klasse die ten grondslag ligt aan veel instanties is de klasse:

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>)  :: m a -> m b        -> m b
  return :: b                -> m b
  fail   :: String           -> m b

  mf >> ma = fm >>= \_ -> ma
  fail s = error s
```

De **do**-notatie biedt syntactische suiker voor aanroepen van deze functies.

Op Monads gebaseerde functies

$$
\begin{aligned}
&\mathit{sequence} \;\; :: \mathit{Monad}\ m \Rightarrow [m\ a] \to m\ [a] \\
&\mathit{sequence} \;\; = \mathit{foldr}\ \mathit{mcons}\ (\mathit{return}\ [\,]) \\
&\quad \textbf{where}\ \mathit{mcons}\ p\ q = p \gg\!= \lambda x \to q \gg\!= \lambda y \to \mathit{return}\ (x : y) \\
&\mathit{sequence\_} \;\; :: \mathit{Monad}\ m \Rightarrow [m\ a] \to m\ () \\
&\mathit{sequence\_} \;\; = \mathit{foldr}\ (\gg)\ (\mathit{return}\ ()) \\
&\mathit{mapM} \;\; :: \mathit{Monad}\ m \Rightarrow (a \to m\ b) \to [a] \to m\ [b] \\
&\mathit{mapM}\ f\ xs = \mathit{sequence}\ (\mathit{map}\ f\ xs) \\
&(=\!\ll) \;\; :: \mathit{Monad}\ m \Rightarrow (a \to m\ b) \to m\ a \to m\ b \\
&f =\!\ll x \;\; = x \gg\!= f
\end{aligned}
$$

IO

In de "geheime" prelude wordt dus het type *IO* een instantie van *Monad* gemaakt. Dit zou als volgt gegaan kunnen zijn:

```
data Buitenwereld = ...    -- een geheim type
data IO a = IO (Buitenwereld→(a, Buitenwereld))
```

IO

In de "geheime" prelude wordt dus het type *IO* een instantie van *Monad* gemaakt. Dit zou als volgt gegaan kunnen zijn:

```
data Buitenwereld = ...    -- een geheim type
data IO a = IO (Buitenwereld→(a, Buitenwereld))
instance Functor IO where
fmap f (IO sea) = IO (λbw → let (v,  nbw) = sea bw
                            in (f v, nwb))
```

IO

In de "geheime" prelude wordt dus het type *IO* een instantie van *Monad* gemaakt. Dit zou als volgt gegaan kunnen zijn:

```
data Buitenwereld = ...    -- een geheim type
data IO a = IO (Buitenwereld→(a, Buitenwereld))
instance Functor IO where
fmap f (IO sea) = IO (λbw → let (v,  nbw) = sea bw
                            in (f v, nwb))
instance Monad IO where
  (IO sea) >>= a2seb = IO (λbw → let (     v,nbw) = sea bw
                                 in a2seb v  nbw)
  return v           = IO (λbw → (v,bw))
```

IO

In de "geheime" prelude wordt dus het type *IO* een instantie van *Monad* gemaakt. Dit zou als volgt gegaan kunnen zijn:

```
data Buitenwereld = ...    -- een geheim type
data IO a = IO (Buitenwereld→(a, Buitenwereld))
instance Functor IO where
fmap f (IO sea) = IO (λbw → let (v,  nbw) = sea bw
                            in (f v, nwb))
instance Monad IO where
  (IO sea) >>= a2seb = IO (λbw → let (     v,nbw) = sea bw
                                 in a2seb v  nbw)
  return v           = IO (λbw → (v,bw))
```

Op deze manier hebben via het type systeem van Haskell een velige interface met de buitenwereld gemaakt.