

Einführung in die funktionale Programmierung

Prof. Dr. Manfred Schmidt-Schauß

Künstliche Intelligenz und Softwaretechnologie

12. November 2007

Überladung: Es gibt mehrere Funktionen mit gleichem Namen.
der Typ der Argumente bestimmt, welche gemeint ist.

$(1; i) + (3; 2i)$ statt $(1, i)$ `complexadd` $(3; 2i)$.

Haskell-Typklassen: zur Überladung von Funktionen
im polymorphen Typsystem.
vordefinierte und benutzerdefinierte

Kombination aus: Typklassen
Milner-(polymorphe) Typisierung
starke Typisierung

Gesucht: Typsystem und Typcheck-Algorithmus
(Typcheck dazu im späteren Kapitel)

- $+$, $*$, $-$, $/$ soll für verschiedene Arten von Zahlen verwendet werden:
Int, Integer, Float, Rational, ...
- Manche Funktionalitäten sind nicht auf allen Typen definierbar;
nur auf einer Menge von vorgegebenen Typen:
z.B.: `(==)`
`show`

Typklasse Analog zu (generischer) Klasse

entspricht einer Menge von Typen zusammen mit
zugehörigen Funktionen bzw. Methoden.

ist eine eigene syntaktische Einheit; mit Namen

Eine neue Typklasse in Haskell wird definiert durch:

```
class newclass (a)
```

Bzw.

```
class <Vorbedingung> => newclass (a)
```

<Vorbedingung>: Klassenbedingungen an die Variable *a*
newclass: neuer Klassenname.
Muss mit Großbuchstabe beginnen

Beispiel

```
class (Eq a, Show a) => Num a
```

<Vorbedingung> ist eine Unterklassenbeziehung von Typklassen
Im allgemeinen sind Typklassen einstellig (nur **nur eine** Variable *a*)

Beispiel

```
class (Eq a, Show a) => Num a
```

Eq a bedeutet:

a gehört zur Typklasse Eq

(Eq a, Show a) => Num a bedeutet:

Nur für Typen a in den Klassen Eq, Show,
wird a auch in Num definiert:

d.h. Num ist **Unterklasse** von Eq, Show.

Bedeutung der Schreibweise:

Eq a => Num a:

Unter der Bedingung Eq a kann man Num a definieren

Eq a => Num a hat die Bedeutung: $\text{Num} \subseteq \text{Eq}$

Deklaration von Typen (von Funktionen):

`class <Vorbedingung> => <Typ>`

Beispiel

Der Typ von `==` ist: `Eq a` \Rightarrow `a -> a -> Bool`

Bedeutung: für alle Typen `a`, die in `Eq` sind,
hat `==` den Typ `a -> a -> Bool`

$t :: \langle \text{Vorbedingung} \rangle \Rightarrow \tau$

Der Ausdruck t hat alle Grundtypen,
die Instanz des Typs τ sind,
wobei die Typklassen-Vorbedingung
beachtet werden muss.

Z.B. $+ :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$

hat die Typen: $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$
 $\text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}$
 $\text{Rational} \rightarrow \text{Rational} \rightarrow \text{Rational}$
...

`s == t`

der **Typchecker** prüft, ob die Typen von s, t in der Klasse `Eq` sind; und ob die Typen gleich sind, bzw. gleich gemacht werden können.

Wenn nein: Ausdruck ist ungetypt

Wenn ja: berechneter Typ kann zur Überladungsauflösung verwendet werden.

Z.B. falls Grundtyp: der richtige Gleichheitstest kann direkt eingesetzt werden

Falls allgemeiner Typ: Einsetzen oder Laufzeit-Typinformation

- In Eq sind alle Typen, für die ein Gleichheitstest implementiert worden ist.
- Nicht sinnvoll ist == z.B. für Funktionen; oder für Datenstrukturen, die Funktionen enthalten

Definition der Typklasse Eq:

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

objektorientierte Sichtweise:

Eq ist eine Klasse,
== und /= sind Methoden
== ist nur Schnittstelle
/= standardmäßig aufbauend auf == definiert
 kann aber überschrieben werden

- Definitionen der Methoden können überschrieben werden in Klassen, die Instanzen sind.
- Deren Typen können nicht überschrieben werden.

- es ist **nicht immer** sinnvoll, Klassen einer objektorientierten Sprache auch als Typklassen in Haskell zu implementieren. Oft ist die richtige Übersetzung ein Datentyp: z.B. Listen.

```
instance Eq Char      where
    x == y           =    fromEnum x == fromEnum y
```

Gleichheit auf Zahlen:

```
instance Eq Int      where (==)    = primEqInt
instance Eq Integer  where (==)    = primEqInteger
instance Ord Int     where compare = primCmpInt
instance Ord Integer where compare = primCmpInteger
```

(Ord wird noch genauer besprochen)

```
instance Eq a => Eq [a]      where
  [] == []                  = True
  [] == (x:xs)              = False
  (x:xs) == []              = False
  (x:xs) == (y:ys)         = x == y && xs == ys
```

- rekursive Definition einer unendlichen Menge von Eq-Typen
- Def. für alle Konstruktormöglichkeiten ([] und :)
- Alternative: Default-Definition: `deriving (Eq,Ord)`.
Hierbei wird automatisch ein Gleichheitstest implementiert, der Strukturgleichheit testet; und Vergleichsfunktionen

Mengen, als Listen implementiert (genauer in progs2) :

```
data Menge a = Set [a]
```

```
Instance Eq a => Eq (Menge a)  where
```

```
    Set xs == Set ys  = subset xs ys && subset ys xs
```

```
subset: Eq a => a -> a -> a
```

```
subset xs ys = all ('elem' ys) xs
```

```
instance (Ord a) => Ord (Menge a)  where
```

```
    Set xs <= Set ys = subset xs ys
```

```
    Set xs >= Set ys = subset ys xs
```

```
    Set xs < Set ys  = subset xs ys && not (subset ys xs)
```

```
    Set xs > Set ys  = subset ys xs && not (subset xs ys)
```

Gleichheit auch für verschachtelte Algebraische Datentypen

Mengen von Mengen

$$\{\{1, 2\}, \{3, 2\}\} == \{\{2, 3, 3\}, \{2, 1\}, \{2, 1\}\}$$

$$\text{Set} [\text{Set}[1,2], \text{Set} [3,2]] == \text{Set} [\text{Set}[2,3,3], \text{Set}[2,1], \text{Set} [2,1]]$$

Mengen von Listen

$$\text{Set} [[1,2], [100..200]] == \text{Set} [[100..200], [100..200], [1,2]]$$

Listen von Mengen

$$[\text{Set}[1,2], \text{Set}[100..200]] == [\text{Set}[100..200], \text{Set}[100..200], \text{Set}[1,2]]$$

$$[\text{Set}[1,2], \text{Set}[100..200]] == [\text{Set}[2,1], \text{Set}[100..200]]$$

```
instance Show a => Show (Menge a) where
  showsPrec p = showMenge
```

```
showMenge (Set [])      = showString "{}"
showMenge (Set (x:xs)) = showChar '{' . shows x . showm xs
  where showm []        = showChar '}'
        showm (x:xs)   = showChar ',' . shows x . showm xs
```

Testen der Mengenimplementierung:

```
*Main> :t Set [[1],[2,3]]
{[1],[2,3]} :: Set [[1],[2,3]] :: forall t. (Num t) => Menge [t]
```

```
*Main> :t Set [Set [1], Set [1,2]]
{{1},{1,2}} :: forall a. (Num a) => Menge (Menge a)
*Main> show (Set [Set [1], Set [1,2]])
"{{1},{1,2}}"
```

Typen mit einer totalen Ordnungsrelation $<$.

```
data Ordering = LT | EQ | GT
    deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)

class (Eq a) => Ord a where
    compare          :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min        :: a -> a -> a

-- Minimal complete definition: (<=) or compare
-- using compare can be more efficient for complex types
compare x y
    | x==y      = EQ
    | x<=y     = LT
    | otherwise = GT
```

Partielle (nicht-totale) Ordnungen:

`compare`; `min` und `max` sind nicht sinnvoll definiert

```
x <= y      = compare x y /= GT
x <  y      = compare x y == LT
x >= y      = compare x y /= LT
x >  y      = compare x y == GT
```

```
max x y
  | x >= y      = x
  | otherwise   = y
min x y
  | x <= y      = x
  | otherwise   = y
```

Die primitive Operation ist `<=`,
die anderen sind darauf aufgebaut.

Zahlen, Character, Tupel und Listen,
und geschachtelte Ord-Datentypen
haben dann die Methode `<=`.

Beispiel `[1,2] < [1,2,3]`,
`[1,2] < [1,3]`,
`(1,'c') < (2,'b')`

Bei Verwendung von `deriving (Eq,Ord)`:
die Datenkonstrukoren werden nach ihrem Index geordnet,
Komponenten lexikographisch.

- Show: alle Typen, die eine Druckmethode haben.
- Read: alle Typen, deren Objekte aus einer Textrepräsentation zu rekonstruieren sind.
- Show und Read Drucken und Parsen sollten invers sein:
 $\text{read}(\text{show}(x)) = x$.

Funktionen (Methoden) für die Klassen Read, Show:

```
shows  :: Show a => a -> ShowS
reads  :: Read a  => ReadS a
show   :: Show a => a  -> String
read   :: Read a => String -> a
type   ReadS a = String -> [(a,String)]
type   ShowS   = String -> String
```

Diese sind teilweise vordefiniert.

Bemerkungen

Die Funktionen `shows` und `reads` dienen der besseren Ressourcennutzung.

Programmierung mittels Funktionskomposition ist umständlich und gewöhnungsbedürftig, aber **effizient**

```
class Show a where
  showsPrec      :: Int -> a -> ShowS
  show           :: a -> String
  showList       :: [a] -> ShowS

  -- Minimal complete definition:
  --      show or showsPrec
  showsPrec _ x s = show x ++ s
  show x         = showsPrec 0 x ""

  showList []      = showString "[]"
  showList (x:xs)  = showChar '[' . shows x . showl xs
                    where showl []      = showChar ']'
                          showl (x:xs) = showChar ',' . shows x .
                                          showl xs
```

Analogie zur multiplen Vererbung:

```
class (Eq a, Show a) => Num a where
  (+), (-), (*)  :: a -> a -> a
  negate        :: a -> a
  abs, signum   :: a -> a
  fromInteger   :: Integer -> a
```

-- Minimal complete definition: All, except negate or (-)

```
x - y          = x + negate y
negate x       = 0 - x
```

```
instance Num Int where
```

```
(+)          = primPlusInt  
(-)          = primMinusInt  
negate       = primNegInt  
(*          = primMulInt  
abs          = absReal  
signum       = signumReal  
fromInteger  = primIntegerToInt
```

Num-Typen müssen die Methoden
von Eq und Show implementiert haben.

Der Ausdruck sei: $(f\ a\ b\ c)$

- Typen der Argumente a, b, c werden vom Typchecker ermittelt
- Damit auch der Typ von f
- Der Compiler analysiert den berechneten Typ
- Falls Grundtyp: Implementierung von f ist bekannt
- Falls Typvariablen, beachte:

Im Haskell-Laufzeitsystem gibt es keine Typinformationen mehr.
`instanceof` gibt es nicht.

Aber: Der Compiler kann ersatzweise dictionary-Parameter einfügen

Beispiel Compiler erkennt die Implementierung

```
let len = length eingabe
    x    = 2*len
in ....
```

- `len:: Int`
- `*:: Num a => a -> a -> a`
- benutzter Instanztyp für `*`: `Int -> Int -> Int`
- Implementierung: Int-Multiplikation
- **Auflösung zur Compilezeit**

Beispiel Compiler weist das Programm wegen Mehrdeutigkeit zurück

In Hugs (alte Version)

ließ sich `[] == []` nicht typisieren,
da `[]::[a]` und das rechte `[]::[b]`
und über `a,b` nichts weiter bekannt ist.

Aber: in `ghci`: Interpreter-Eingabe `[] == []` ist typisierbar
Offenbar wird erkannt, dass `a,b` redundant sind.

Beispiel zu Auflösung der Überladung

`[] == []`

`== :: Eq a => a -> a -> Bool`

Da `[] :: [b]`, erhält man:

`== :: Eq b => [b] -> [b] -> Bool`

Da *b* redundant ist,

kann man jede Implementierung von `==` auf Listen nehmen.

Fall: Eine überladene Funktionen hat keinen eindeutigen Typ:
Aber der Compiler erkennt:
ist dynamisch mittels late binding auflösbar

Vorgehen des Compilers / Interpreters

- baut ein „Dictionary“ auf,
als normale Haskell-Datenstruktur
als Baum-Abbildung der Typklassenhierarchie und der „Methoden“
Blätter sind **Methodenimplementierung**
Knoten entsprechen den **Fallunterscheidungen** über die Typen
- fügt zur Funktion einen **Dictionary-Parameter** hinzu
- Auch andere definierte Funktionen bekommen teilweise einen **Dictionary-Parameter**
- Zur Laufzeit: Ermittlung der richtigen Implementierung

```
f x y u v = if x == y then u else v
  :: forall a t. (Eq a) => a -> a -> t -> t -> t
```

Eine mögliche Transformation:

```
f dict x y u v = if eqswitch dict x y then u else v
```

Kompiler ermittelt bei Benutzung von `f`
anhand des berechneten Typs:
welcher Dictionary-Parameter zu benutzen ist.

Beachte

Ein Dictionary kann rekursiv aufgebaut sein,

Es muss aber endlich sein

Neue Art des Typfehlers: „unendliches Dictionary“

Datentypen, deren Elemente sich (vorwärts, und rückwärts) aufzählen lassen:

Beispiele: nichtnegative ganze Zahlen;
die Zeichen eines Alphabets,
Wochentage
Farben, (wenn man Ordnung festlegt)
.....

überladene Methoden von Enum

<code>enumFrom</code>	<code>[2..]</code>	ergibt <code>[2,3,4,5,...]</code>
<code>enumFromTo</code>	<code>['a'..'z']</code>	ergibt <code>['a','b','c','d',...]</code>
<code>enumFromThen</code>	<code>[3,5..]</code>	ergibt <code>[3,5,7,...]</code>
<code>enumFromThenTo</code>	<code>['1','2'..'z']</code>	ergibt <code>['1','2','3',..., 'z']</code>

```
class Enum a where
  toEnum          :: Int -> a
  fromEnum        :: a -> Int
  enumFrom        :: a -> [a]           -- [n..]
  enumFromThen    :: a -> a -> [a]     -- [n,m..]
  enumFromTo      :: a -> a -> [a]     -- [n..m]
  enumFromThenTo  :: a -> a -> a -> [a] -- [n,n'..m]

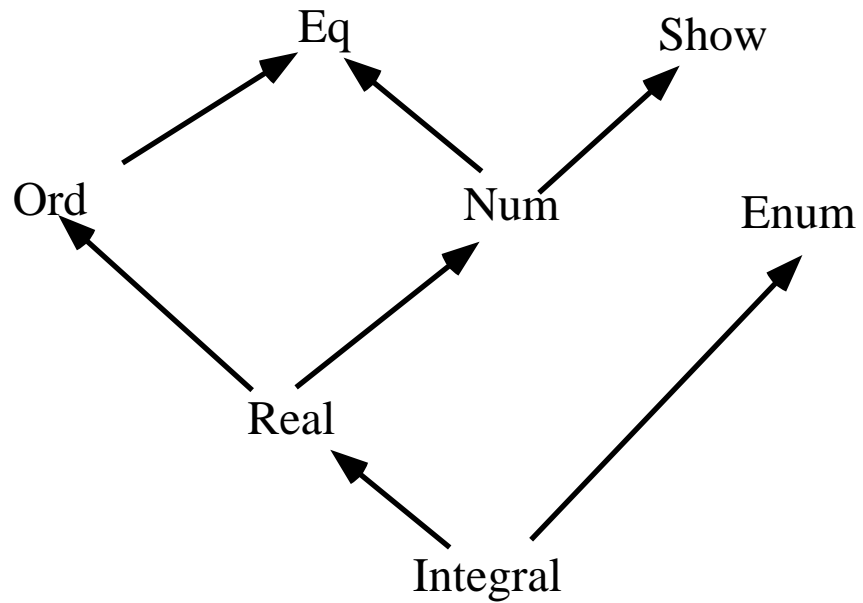
  enumFromTo x y      = map toEnum [fromEnum x .. fromEnum y ]
  enumFromThenTo x y z = map toEnum [fromEnum x,
                                     fromEnum y .. fromEnum z ]

succ, pred :: Enum a => a -> a
succ       = toEnum . (1+)      . fromEnum
pred      = toEnum . subtract 1 . fromEnum
```

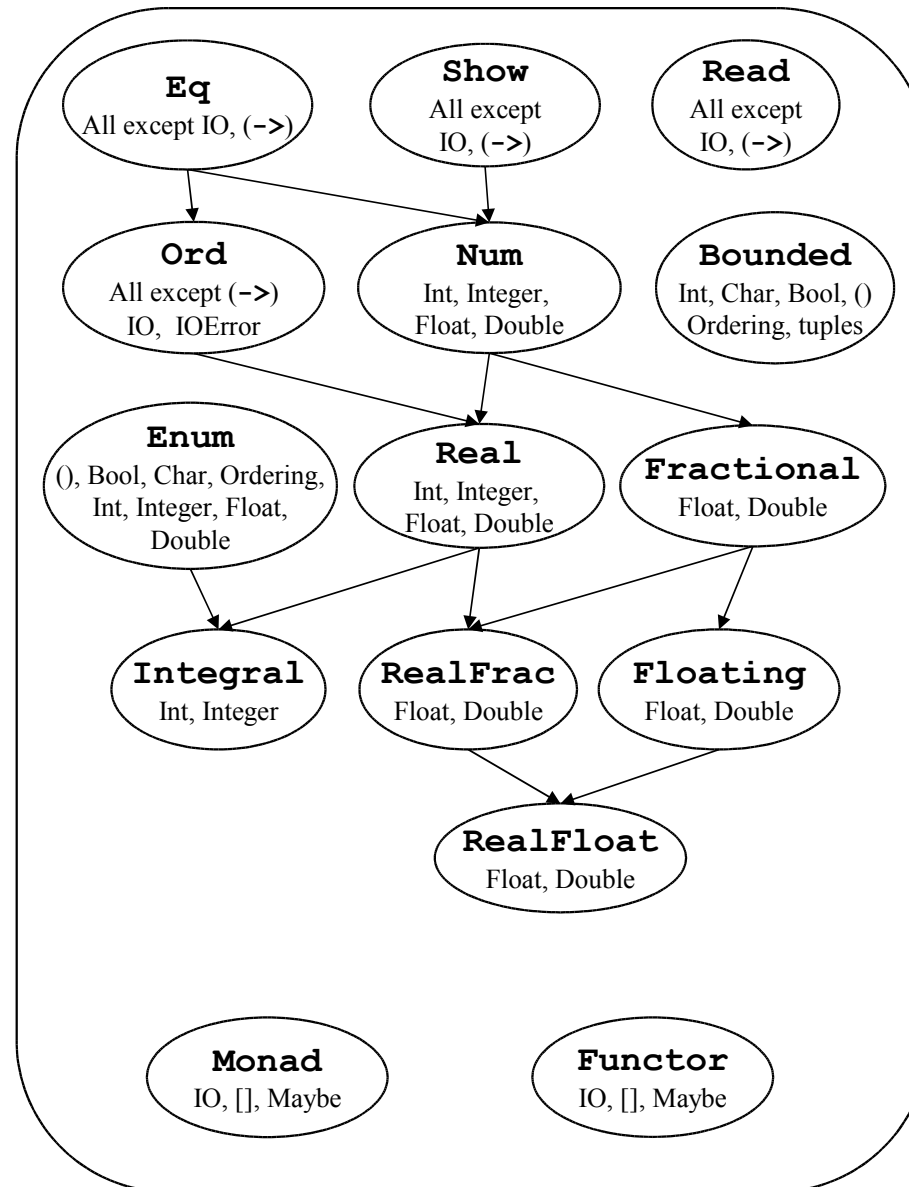
Beispiel: Auszug aus Hugs-Prelude (2)

```
instance Enum Char where
  toEnum          = primIntToChar
  fromEnum        = primCharToInt
  enumFrom c      = map toEnum
                    [fromEnum c .. fromEnum (maxBound::Char)]
  enumFromThen c d =
    map toEnum [fromEnum c, fromEnum d .. fromEnum (lastChar::Char)]
    where lastChar = if d < c then minBound else maxBound
```

Typklassenhierarchie (Ausschnitt aus den vordefinierten Typklassen)



Typklassenhierarchie aus www.haskell.org



Die Typklasse `Finite`

einzigste Methode: `members`, die alle Elemente des Typs als Liste enthält.

```
class Finite a where members :: [a]
```

```
instance Finite Bool where members = [True, False]
```

```
instance (Finite a, Finite b) => Finite (a, b) where  
  members = [(x,y) | x <- members, y <- members]
```

`members`: ist eine Konstante, die abhängig von ihrem Typkontext ihren Wert ändert.

Damit erhält man z.B.

```
members :: Bool -> [ True, False]
```

```
length (members :: [(Bool, Bool), (Bool, Bool)]) ---> 16
```

Rekursive Definition für Listen:

```
instance (Finite a) => Finite [a] where
    members = (powerset (members))
```

```
members :: [[Bool]]
[[True,False], [True], [False], []]
```

```
members :: [[[Bool]]]
[[[True,False], [True], [False], []],
 [ [True,False], [True], [False] ],
 [ [True,False], [True], [] ],
 ....
]
```

Für Listen vom Typ `Num a => [a] => a`

Der folgende Versuch scheiterte in früherem Haskell:

```
summe [] = 0
summe (x:xs) = x+summe xs
```

Grund: die Zahl 0 hatte beim Einlesen den festen Typ `Int`

Aktuelle Haskell-Versionen:

0 wird als `fromInteger 0` eingelesen

Einfache Definition, richtige Typen, richtige Funktionalität:

```
allgemeine_summe :: Num a => [a] -> a
allgemeine_summe [] = 0
allgemeine_summe (x:xs) = x + allgemeine_summe xs
```

(Effizienter ist die iterative Variante mit Akkumulator)

Abstraktion der map-Funktionalität

In der aktuellen Haskell-Version: `fmap` statt `map`

```
class Functor f where
    fmap :: (a -> b) -> (f a -> f b)
```

```
data Maybe a = Just a | Nothing
    deriving (Eq, Ord, Read, Show)
```

```
instance Functor Maybe where
    fmap f Nothing = Nothing
    fmap f (Just x) = Just (f x)
```

```
instance Functor [] where
    -- [] bedeutet hier List als Typkonstruktor, nicht nil
    fmap = map
```

Sinnvoll ist es, auch andere Datenstrukturen, für die man ein `map` definieren kann, zu Instanzen von `Functor` zu machen.

```
data Baum a = Blatt a | Knoten (Baum a) (Baum a)
    deriving (Eq, Show)
```

```
instance Functor Baum where
    fmap = baummap
```

```
baummap f (Blatt a) = Blatt (f a)
baummap f (Knoten x y) = Knoten (baummap f x) (baummap f y)
```

```
test = fmap (\x -> 2*x) (Knoten (Blatt 1) (Blatt 2))
```

In späteren Kapitel sind einige Hinweise dazu

Möglichkeiten:

- Mehr als eine Typvariable in der Definition einer Typklasse
- komplexer strukturierte Typen.

Problem der Verallgemeinerungen:

Gibt es einen entscheidbaren Typcheckalgorithmus?

Pragmatischer: gibt es einen brauchbaren Typcheck?

Eine Antwort: Für eine Kombination von Multiparameter-Typklassen mit “functional dependencies und undecidable instances” ist der Typcheck unentscheidbar

Allgemeineres Typsystem: abhängige Typen (dependent types)

sehr ausdrucksstark, allerdings unentscheidbar

Typ der Objekten abhängig von Attribut-Werten

Beispiel

Matrizen: Typbeschreibung enthält Anzahl der Zeilen und Spalten
Spalten (Zeilen)-Anzahl nur dynamisch bekannt (nicht statisch)

Einfachster, sinnvoller Fall in Haskell sind **Tupel**,
In vollem Umfang in einem dependent-type System implementierbar.

Erinnerung: man bräuchte unendlich viele Funktionen,
um alle Tupelfunktionalitäten in Haskell vorzudefinieren,
Z.B.: **extrahiere i -tes Element eines n -Tupels**

Es gibt prototypische Implementierungen zu dependent-types
in experimentellen funktionalen Programmiersprachen

Haskell Erweiterungen: Tupel mittels Phantom-Typen