IDRIS Workshop - Notizen

Carsten König

Devopenspace Leipzig 2016

Contents

Chapter 1

Basics

Links

- IDRIS
- Basiert auf [Edwin Brady:Type-Driven Development with Idris
- Online Doc

Idris

- Interpreter aufrufen
- `:t`
- `:doc`
- `:browse`
- `:apropos`
- `:m`

nummerische Typen

- `Int` = Ganzzahlen mit Vorzeichen (mindestens 31bit) (`0,1,-10,...`)
- `Integer` unbegrenzte Ganzzahlen
- `Nat` unbegrenzte natürliche Zahlen (`>= 0` <- mehr davon später)
- `Double` Fließkommazahl (`3.14`)

Beispiel

```
12 + 5 * 6 ;; 42 : Integer
2 * 3.14   ;; 6.28 : Double
```

wie gebe ich einen Typ an?

Was wenn ich `2` als `Double` haben möchte?

```
the Double 2
```

Casts

```
cast "21" * 2
```

Frage

Wie werden die einzelnen Typen hergeleitet?

Antwort - `(*)` will zwei vom gleichen Typ - das Literal `2` wird ohne weitere Angabe zu `Integer` - damit muss `cast` einen `Integer` liefern

---

Strings

- `Char` Beispiel `'c'`
- `String` Beispiel `"Hallo"`
- Suche nach Funktionen mit ":apropos String"

  Vorsicht: `String` is kein `List String`

aber das geht mit `unpack` und `pack`:

```
> unpack "Hello"

> pack it
```

- Aneinander hängen mit `++`

Booleans

- `Bool` mit `True` und `False`
- `||`, `&&`, `==` und `/=`
- `if ... then ... else ...`

Funktionen

Einfache Funktion

```
hallo : String -> String
hallo name = "Hallo " ++ name
```

- immer `[name] : [Typ]`
- Vorsicht: Typen sind auch Typen (`> :t Type`)
- Signatur `quellTyp -> zielTyp`
- Definition/Body

Lambdas

```
f : Int -> Int
f = \n => n+1
```

Mehrere Parameter

```
addieren : Int -> Int -> Int
addieren a b = a + b
```

- eigentlich `Int -> (Int -> Int)`
- Currying

```
addieren' : Int -> (Int -> Int)
addieren' a = \b => a + b
```

partial Applikation

```
add10 : Int -> Int
add10 = addieren 10
```

> alles klar hier?

---

generische Funktionen

```
identität : ty -> ty
identität val = val
```

eigentlich sogar

```
identität : {ty : Type} -> ty -> ty
identität val = val
```

zeige die *Impliziten Werte* (in Emacs)

Löcher

```
identität : ty -> ty
identität val = ?val
```

```
identitaet : {ty : Type} -> ty -> ty
identitaet {ty = typ} val = val
```

Zusammengesetzte Datentypen

Liste

```
zahlen : List Nat
zahlen = [1,2,3,4,5]

zahlen' : List Nat
zahlen' = [1..5]

namen : List String
namen = "Marie" :: "Carsten" :: []
```

Tuple

```
person : (String, Nat)
person : ("Max", 40)
```

Union Type

```
data Farbe
  = Rot
  | Blau

data Ergebnis
  = Fehler String
  | Erfolg Int
```

GADT

```
-- GADT
data Ergebnis' : Type where
  Fehler' : String -> Ergebnis'
  Erfolg' : Int -> Ergebnis'

data Expression : Type -> Type where
  Falsch : Expression Bool
  Wahl : Expression Bool
  Zahl : Nat -> Expression Nat
  Plus : Expression Nat -> Expression Nat -> Expression Nat
  Gleich : Expression Nat -> Expression Nat -> Expression Bool
  Falls : Expression Bool ->
          Expression a -> Expression a ->
          Expression a
```

Patter Matching

Listen

```
istLeer : List a -> Bool
istLeer [] = True
istLeer (x :: xs) = False
```

Tupel

```
name : (String, Nat) -> String
name (n, _) = n
```

GADT

```
eval : Expression a -> a
eval Falsch = False
eval Wahr = True
eval (Zahl k) = k
eval (Plus x y) = eval x + eval y
```

```
eval (Gleich x y) = eval x == eval y
eval (Falls b t e) = if (eval b) then eval t else eval e
```

Übung

Vervollständige

```
concat : List a -> List a -> List a
concat xs ys = ?concat
```

Chapter 2

Typ Level Funktionen

Typ Synonmye

```
Zahl : Type
Zahl = Nat
```

Funktionen

```
NatOderString : Bool -> Type
NatOderString False = Nat
NatOderString True = String
```

in Signatur

```
natOrString : (b:Bool) -> NatOderString b
natOrString False = 42
natOrString True = "Frage"

toString : (b:Bool) -> NatOderString b -> String
toString False n = show n
toString True s = s
```

- `Type` verschwindet beim Kompilieren (also auch die Funktionen)
- Kompiler wertet Typ-Level Funktionen nur aus, wenn Sie total sind

Chapter 3

Vektoren

Zeige eine Beispielimplementation:

```
data Vektor : (n:Nat) -> (a:Type) -> Type where
  Nil : Vektor 0 a
  (::) : a -> Vektor n a -> Vektor (S n) a


concat : Vektor n a -> Vektor m a -> Vektor (n+m) a
concat [] ys = ys
concat (x :: xs) ys = x :: concat xs ys
```

Beispiele

```
import Data.Vect

fourInts : Vect 4 Int
fourInts = [0, 1, 2, 3]

sixInts : Vect 6 Int
sixInts = [4, 5, 6, 7, 8, 9]

tenInts : Vect 10 Int
tenInts = fourInts ++ sixInts
```

Übung

Implementiere

- `vecMap : (a -> b) -> Vect n a -> Vect n b`
- `vecLength : Vect n a -> Nat`
- `vecTail : ???`

Lösung

```
vecMap : (a -> b) -> Vect n a -> Vect n b
vecMap f [] = []
vecMap f (x :: xs) = f x :: vecMap f xs

vecLength : Vect n a -> Nat
vecLength {n} _ = n


-- Wie könnte ein typsicherere Tail-Funktion aussehen?
vecTail : Vect (S n) a -> Vect n a
vecTail (x :: xs) = xs
```

Matrizen

Ziel:

```
Matrix : (Nat, Nat) -> Type -> Type
Matrix (n,m) a = Vect n (Vect m a)

matTranspose : Matrix (n,m) a -> Matrix (m,n) a
```

Lösung

```
mat0 : Matrix (n,0) a
mat0 {n} = replicate n []
```

```
matTranspose : Matrix (n,m) a -> Matrix (m,n) a
matTranspose [] = mat0
matTranspose (x :: xs) =
  let xs' = transpose xs
  in zipWith (::) x (transpose xs)
```

Übung

Implementiere

- `matMult : Num a => Matrix (n,m) a -> Matrix (m,o) a -> Matrix (n,o) a`

Hinweise

- Funktioniert über "Zeile MAL Spalte"
- MAL ist dabei das Skalarprodukt (implementieren)
- Schleifen für Zeile und Spalte, können über `map` implementiert werden

```
scalProd : Num a => Vect n a -> Vect n a -> a
scalProd xs ys = sum (zipWith (*) xs ys)

matMult : Num a => Matrix (n,m) a ->
                   Matrix (m,o) a ->
                   Matrix (n,o) a
matMult xs ys =
  let ys' = matTranspose ys
  in map (\x => map (scalProd x) ys') xs
```

Beispiel

```
> matMult [[1,2],[3,4],[5,6]] [[7,8,9,10],[11,12,13,14]]
[ [29, 32, 35, 38]
, [65, 72, 79, 86]
, [101, 112, 123, 134]]
```

Fin

Wollen:

```
index : Nat -> Vect n a -> a
```

Wo sind die Probleme?

- Index out of Bound

Übung: Implementiere das

```
tryIndex : Nat -> Vect n a -> Maybe a
```

Wäre schöner: Index funktion die garantiert ein Ergebnis liefert

---

Dafür `Fin`

```
data Kleiner : Nat -> Type where
  E0 : Kleiner (S n) -- für alle n:Nat gilt 0 ist kleiner n+1
  ES : Kleiner n ->
       Kleiner (S n) -- falls x kleiner n ist x+1 kleiner n+1


zweiKleiner4 : Kleiner 4
zweiKleiner4 = ES (ES E0)
```

jetzt wollen wir

```
index' : Fin n -> Vect n a -> a
```

Lösung

```
index' : Fin n -> Vect n a -> a
index' FZ (a :: _) = a
index' (FS ind) (_ :: as) = index' ind as
```

Übung

`take : Int -> List a -> List a`

wohl bekannt, wie kann das für `Vect` aussehen?

Lösung

```
vectTake : (n:Nat) -> Vect (n+m) a -> Vect n a
vectTake Z xs = []
vectTake (S k) (x :: xs) = x :: vectTake k xs
```

Chapter 4

Dependent Pairs

mit `n ** Abhängigkeit(n)`

Beispiel

Implementiere `filter` für Vektoren

```
filter' : (a -> Bool) -> Vect n a -> (m ** Vect m a)
filter' pred [] = (0 ** [])
filter' pred (x :: xs) =
  let (_ ** rek) = filter' pred xs
  in if pred x then (_ ** x::rek) else (_ ** rek)
```

Eingabe: `filter' (\x => (xmod2) == 0) [1,2,3,4,5]`

Chapter 5

IO

Ähnlich wie in Haskell über `IO resType`

```
main : IO ()
main = do
    putStr "Enter your name: "
    x <- getLine
    putStrLn ("Halo " ++ x ++ "!")
```

- `IO a` sagt: ich bin eine Aktion mit Seiteneffekten, die nach Abschluss ein `a` zurückgibt
- `putStr str`: schreibe String `str` in `stdout`
- `putStrLn` schreibe String `(str + '\n')` in `stdout`
- `getLine` liest alles bis zur nächsten `'\n'` von `stdin`

in der REPL über

```
:exec -- ruft die Main auf
:x aktion -- führt eine IO-Aktion aus
```

Monaden / Bind

- `(>>=) :  IO a -> (a -> IO b) -> IO b` und
- `do` erklären falls nötig

Übung

Schreib eine Aktion, dass zwei Zahlen eingeben lässt und dann die Summe ausgibt

Hinweis

Unter Windows rennt man in ziemlich blöde Buffering Probleme

Ist unter Windows/Emacs ziemlich lästig - unbedingt eine `main : IO ()` verwenden und `:exec` benutzen (`:x` geht gar nicht)

Atom: sorry

besser:

```
idris InputOutput.idr --exec plusIO
```

oder:

```
idris InputOutput.idr -o InputOutput.exe
InputOutput.exe
```

Lösung

```
module Main

plusIO : IO Int
plusIO = do
  putStr "Zahl 1: "
  zahl1 <- getLine
  putStr "Zahl 2: "
  zahl2 <- getLine
  pure $ cast zahl1 + cast zahl2

main : IO ()
main = do
  erg <- plusIO
  putStrLn $ show erg
```

> Was passiert wenn keine Zahl eingegegeben wird

(Doku durchsuchen)

eine `Maybe Zahl` auslesen

- `unpack` - String in Liste aus Chars
- `all` prüft Eigenschaft auf alle Elemente
- `isDigit` prüft auf Ziffer

```
readNumber : IO (Maybe Nat)
readNumber = do
  input <- getLine
  if all isDigit (unpack input) then
    pure (Just (cast input))
  else
    pure Nothing
```

Übung

Guess Number Spiel implementieren

(siehe Guess-Nr Projekt)

Lösung

```
module Main

import System

random : IO Integer
random = time

readNumber : IO (Maybe Nat)
readNumber = do
  input <- getLine
  if all isDigit (unpack input) then
    pure (Just (cast input))
  else
    pure Nothing

rate : Nat -> Nat -> IO ()
rate ziel Z = putStrLn "Leider Verloren!"
rate ziel (S versuche) = do
  putStrLn ("Du hast noch " ++ show (S versuche) ++ " Versuche")
  putStr "Zahl? "
```

```
  input <- readNumber
  case input of
    Nothing => rate ziel (S versuche)
    Just zahl =>
    if zahl == ziel then do
      putStrLn "Du hast es geschaft"
    else if zahl <= ziel then do
      putStrLn "Deine Zahl ist zu klein"
      rate ziel versuche
    else do
      putStrLn "Deine Zahl ist zu groß"
      rate ziel versuche

zahlZwischen : Integer -> Integer -> IO Nat
zahlZwischen von bis =
  do
    zufall <- random
    pure (cast (von + zufall `mod` range))
  where range = bis - von

main : IO ()
main = do
  ziel <- zahlZwischen 1 100
  rate ziel 5
```

Chapter 6

Implementieren PrintF

Aufwärmen

Funktion mit var. Argumentanzahl

```
NNatsFun : Nat -> Type
NNatsFun Z = Nat
NNatsFun (S k) = Nat -> NNatsFun k

adder : (n:Nat) -> Nat -> NNatsFun n
adder Z acc = acc
adder (S k) acc = \n => adder k (acc+n)
```

Übung (* * *)

Schaft ihr das ohne Acc?

Lösung

```
NNatsFun : Nat -> Type
NNatsFun Z = Nat
NNatsFun (S k) = Nat -> NNatsFun k

plusN : Nat -> NNatsFun k -> NNatsFun k
plusN {k = Z} n m = n + m
plusN {k = (S k)} n f = \x => f (x+n)
```

```
adder : (n:Nat) -> NNatsFun n
adder Z = 0
adder (S k) = \n => plusN n (adder k)
```

PrintF

Repräsentation des Format-Strings (Beispiel: `Hallo %s Du bist %d`)

- macht es einfacher Funktionen zu schreiben (case split, ...)
- sagt mehr aus als `String`

```
data Format
     = Num Format
     | Str Format
     | Lit String Format
     | End
```

Im Beispiel ist also:

```
Lit "Hallo " (Str (Lit " Du bist " (Num End)))
```

Wie gerade: Funktionstyp aus dem Format berechnen:

Übung

Wie oben mit adder:

```
PrintfType : Format -> Type
```

Lösung

```
PrintfType : Format -> Type
PrintfType (Num rest) = Integer -> PrintfType rest
PrintfType (Str rest) = String -> PrintfType rest
PrintfType (Lit _ rest) = PrintfType rest
PrintfType End = String
```

Übung

```
printfToString : (format : Format) ->
                 (acc : String) ->
                 PrintfType format
```

Lösung

```
printfToString : (format : Format) ->
                 (acc : String) ->
                 PrintfType format
printfToString (Num rest) acc =
    \num => printfToString rest (acc ++ show num)
printfToString (Str rest) acc =
    \str => printfToString rest (acc ++ str)
printfToString (Lit out rest) acc =
    printfToString rest (acc ++ out)
printfToString End acc = acc
```

Müssen einen String parsen:

Hinweis: `strCons : Char -> String -> String`

```
parseStringToFormat : String -> Format
parseStringToFormat s = parseChars (unpack s)
  where
    parseChars : List Char -> Format
    parseChars [] = End
    parseChars ('%' :: 'd' :: rest) = Num (parseChars rest)
    parseChars ('%' :: 's' :: rest) = Str (parseChars rest)
    parseChars (c::cs) =
      case parseChars cs of
        Lit out rest => Lit (c `strCons` out) rest
        other => Lit (pack [c]) other
```

Fertig stellen:

```
printf : (format : String) ->
         PrintfType (parseStringToFormat format)
printf format = printfToString _ ""
```

der _ dort funktioniert, weil das Format aus der Rückgabe klar ist!

Chapter 7

Type-Level Gleichheit

Beispiel

Führe vor, wie das auf Probleme stößt:

```
vecReverse : Vect n a -> Vect n a
```

Gleichheit auf Typebene über einen Datentyp

```
infixl 2 ===
data (===) : { ty : Type } -> (a : ty) -> (b : ty) -> Type where
  Gleich : a === a

> the (2 === 2) Gleich -- ok
> the (2 === 1+1) Gleich -- ok
> the (2 === 3) Gleich -- nicht ok
```

gibt es glücklicherweise schon als `=` mit `Refl` - Typ! (zeige Doc)

Übung

Implementiere

```
gleicheZahlen : (a : Nat) -> (b : Nat) -> Maybe (a = b)
```

Hinweis: `cong`

Lösung

```
gleicheZahlen : (a : Nat) -> (b : Nat) -> Maybe (a = b)
gleicheZahlen Z Z = Just Refl
gleicheZahlen Z (S k) = Nothing
gleicheZahlen (S k) Z = Nothing
gleicheZahlen (S k) (S j) =
  case gleicheZahlen k j of
    Nothing => Nothing
    Just prf => Just (cong prf)
```

Übung

"Beweise":

```
plus1IstSucc : (n:Nat) -> S n = n+1
```

Lösung

```
plus1IstSucc : (n:Nat) -> S n = n+1
plus1IstSucc Z = Refl
plus1IstSucc (S k) = cong (plus1IstSucc k)
```

Lösung des Beispiels

Einführung Cong

```
vecReverse : Vect n a -> Vect n a
vecReverse [] = []
vecReverse {n = S k} (x :: xs) =
  let rev = vecReverse xs ++ [x]
  in rewrite (plus1IstSucc k) in rev
```

Übung

Definiere einen Datentyp `DreiGleich` der angibt, dass 3 Werte gleich sind

```
data DreiGleich : .... -> Type where
```

implementiere damit die Verallgemeinerung von `cong3`

Lösung

```
data DreiGleich : a -> b -> c -> Type where
  Refl3 : DreiGleich x x x

cong3 : { a,b,c : ty } ->{ f : ty -> ty' } ->
        DreiGleich a b c -> DreiGleich (f a) (f b) (f c)
cong3 Refl3 = Refl3

alleGleichS : {x,y,z : Nat} ->
              DreiGleich x y z ->
              DreiGleich (S x) (S y) (S z)
alleGleichS prf = cong3 prf
```

Entscheidbarkeit

Wir können **aussagen** dass zwei Werte **gleich** sind - was aber, wenn wir **garantieren** wollen, dass sie **nicht gleich** sind?

Wir brauchen irgendwie eine Aussage, dass `x = y` *unmöglich* ist.

Dafür nutzen wir *Void* (einen Datentyp ohne Wert)

```
data Void
```

> Wenn eine Funktion `Void` liefert, kann das nur heißen, dass es nicht möglich ist ihre Eingaben zu konstruieren!

Erkläre ein wenig Curry-Howard

Beispiel

```
unmoeglich : 2+2 = 5 -> Void
unmoeglich Refl impossible
```

Idris bemerkt, dass da was nicht stimmt

Aus einem `Void` Wert (sic) kann man mit `void` jeden Wert generieren!

Typ für Entscheidbarkeit

```
data Entscheidbar : (behauptung:Type) -> Type where
  Ja   : (beweis : behauptung) ->
         Entscheidbar behauptung
  Nein : (widerspruch : behauptung -> Void) ->
         Entscheidbar behauptung
```

Beispiel

```
sindGleich : (n : Nat) -> (m : Nat) -> Entscheidbar (n = m)
sindGleich Z Z = Ja Refl
sindGleich Z (S k) = Nein (NullUngleichNachfolger k)
  where
    NullUngleichNachfolger : Nat -> (Z = S k) -> Void
    NullUngleichNachfolger _ Refl impossible
sindGleich (S k) Z = Nein (NachfolgerUngleichNull k)
 where
   NachfolgerUngleichNull : Nat -> (S k = Z) -> Void
   NachfolgerUngleichNull _ Refl impossible
sindGleich (S k) (S j) =
  case sindGleich k j of
    Ja prf => Ja (cong prf)
    Nein wid => Nein (auchNichtGleich wid)
  where
    auchNichtGleich : (a = b -> Void) -> (S a = S b) -> Void
    auchNichtGleich wid Refl = wid Refl
```

gibt es schon al `Dec`, `Yes`, `No`

Hinweis es gibt ein `decEq` auf (über ein Interface ... im Doc zeigen)

Chapter 8

Typen mit internen Kontrakten

Übung

Implementiere ein `entferne : Eq a => (x : a) -> Vect n a -> ?`

Lösung

```
entferne : Eq a => (x : a) -> Vect n a -> (m ** Vect m a)
entferne x [] = (0 ** [])
entferne x (y :: xs) =
  if x == y then (_ ** xs)
  else
    let (_ ** rest) = entferne x xs
    in (_ ** y :: rest)
```

Probleme:

- was wenn mehrere Elemente (wo entfernen)
- Rückgabe unschön

Besser:

Wenn wir schon wissen, dass `x` in `xs` ist, könnten wir etwas wie

```
entferne : (x : a) -> Vect (S n) a -> Vect n a
```

schreiben!

Typ dafür!

Definiere

```
data IstDrin : a -> Vect n a -> Type where
  Hier : IstDrin a (a :: xs)
  Dort : IstDrin a xs -> IstDrin a (y :: xs)
```

Übung

Implementiere

```
entferne' : (x : a) -> (xs : Vect (S n) a) ->
            IstDrin x xs -> Vect n a
```

Lösung

```
entferne' : (x : a) -> (xs : Vect (S n) a) ->
            IstDrin x xs -> Vect n a
entferne' x (x :: ys) Hier = ys
entferne' x (y :: (x :: xs)) (Dort Hier) =
    y :: xs
entferne' x (y :: (z :: xs)) (Dort (Dort w)) =
    y :: entferne' x (z::xs) (Dort w)
```

Automatische implizite Argumente

```
entferneAuto : (x : a) -> (xs : Vect (S n) a) ->
               {auto p : IstDrin x xs} -> Vect n a
entferneAuto x (x :: ys) { p = Hier } = ys
entferneAuto x (y :: (x :: xs)) { p = Dort Hier } =
    y :: xs
entferneAuto x (y :: (z :: xs)) { p = Dort (Dort prf) } =
    y :: entferneAuto x (z::xs) {p = Dort prf}
```

Eingebaut gibt es `IstDrin` schon als `Elem` mit `Here` und `There`

Übung

Implementiere

```
istDrin : DecEq a => (x : a) -> (xs : Vect n a) ->
          Dec (IstDrin x xs)
```

Lösung

```
istDrin : DecEq a => (x : a) -> (xs : Vect n a) ->
          Dec (IstDrin x xs)
istDrin x [] = No nichtInLeer
  where
    nichtInLeer : IstDrin x [] -> Void
    nichtInLeer Hier impossible
    nichtInLeer (Dort _) impossible
istDrin x (y :: xs) =
  case decEq x y of
    Yes Refl => Yes Hier
    No notHere =>
      case istDrin x xs of
        Yes dort => Yes (Dort dort)
        No nichtDort =>
          No (\doch =>
            case doch of
              Hier => notHere Refl
              Dort d => nichtDort d)
```

Projekt

```
module Baum

-- ein Baum ist entweder ein leeres Blatt
-- oder ein Ast mit einem Wert und zwei Unterbäumen
-- überlege wie der entsprechende Datentyp aussehen könnte

data Baum : (a : Type) -> Type where
  Blatt : Baum a
  Ast : (wert : a) ->
        (links : Baum a) -> (rechts : Baum a) ->
        Baum a

-- ähnlich `Elem` für den Vektor wollen wir eine Datenstruktur
-- die "beweist", dass ein Wert in einem Baum ist, indem der Weg
-- dorthin aufgezeigt wird
-- Wie kann das ausehen?

data PfadZu : (wert : a) -> (baum : Baum a) -> Type where
  Hier : PfadZu wert (Ast wert l r)
  IstLinks : PfadZu wert links -> PfadZu wert (Ast y links r)
  IstRechts : PfadZu wert rechts -> PfadZu wert (Ast y l rechts)

-- jetzt müssen wir noch entscheiden,
-- ob wir einen Weg finden können

nichtImBlatt : PfadZu zu Blatt -> Void
nichtImBlatt Hier impossible
nichtImBlatt (IstLinks _) impossible


nowhere : (notRechts : PfadZu zu rechts -> Void) ->
          (notLinks : PfadZu zu links -> Void) ->
          (notHere : (zu = wert) -> Void) ->
          PfadZu zu (Ast wert links rechts) -> Void
nowhere notRechts notLinks notHere Hier = notHere Refl
nowhere notRechts notLinks notHere (IstLinks l) = notLinks l
nowhere notRechts notLinks notHere (IstRechts r) = notRechts r


gibtEsPfad : DecEq a => (zu : a) -> (baum : Baum a) ->
             Dec (PfadZu zu baum)
gibtEsPfad zu Blatt = No nichtImBlatt
```

```
gibtEsPfad zu (Ast wert links rechts) =
  case decEq zu wert of
    Yes Refl => Yes Hier
    No notHere =>
      case gibtEsPfad zu links of
        Yes links => Yes (IstLinks links)
        No notLinks =>
          case gibtEsPfad zu rechts of
            Yes rechts => Yes (IstRechts rechts)
            No notRechts => No (nowhere notRechts notLinks notHere)


----------------------------------------------------------------------
-- Wenn alles passt sollte
-- `zu0 = Yes (IstLinks (IstRechts Hier))`
-- sein!

beispiel : Baum Int
beispiel = Ast 2 (Ast 1 Blatt (Ast 0 Blatt Blatt))
                 (Ast 3 Blatt Blatt)

zu0 : Dec (PfadZu 0 Baum.beispiel)
zu0 = gibtEsPfad _ _
```

Chapter 9

Guessing Game

State

```
data GameState  : (guessesRemaining : Nat) ->
                  (letters : Nat) ->
                  Type where
    MkGameState : (word : String) ->
                  (missing : Vect letter Char) ->
                  GameState guessesRemaining letters
```

Gewonnen / Verloren

```
data Finished : Type where
  Lost : (game : GameState 0 (S letters)) -> Finished
  Won : (game : GameState (S guesses) 0) -> Finished
```

Spiel - Funktion

```
game : GameState (S guesses) (S letters) -> IO Finished
```

Spielereingabe prüfen

nur ein einzelner Buchstabe ist gültig:

```
data ValidInput : List Char -> Type where
  Letter : (c : Char) -> ValidInput [c]
```

Übung

implementiere

```
isValidInput : (cs : List Char) -> Dec (ValidInput cs)
```

---

damit

```
isValidString : (s : String) -> Dec (ValidInput (unpack s))
isValidString s = isValidInput _
```

Übung

Implementiere

```
readGuess : IO (c ** ValidInput c)
```

Soll nach einem Buchstaben fragen, und die Eingabe (in Großbuchstaben `toUpper` prüfen.

- Ist sie ok soll die Eingabe und der "Beweis" zurückgegeben werden
- Sonst soll nach Fehlermeldung erneut gefragt werden

Lösung

```
readGuess : IO (c ** ValidInput c)
readGuess = do
  putStrLn "Buchstabe? "
  input <- getLine
  case isValidString (toUpper input) of
```

```
    Yes prf => pure (_ ** prf)
    No _ => do
      print "ungültige Eingabe"
      readGuess
```

Verarbeitung

```
processGuess : (letter : Char) ->
               GameState (S guesses) (S letters) ->
               Either (GameState guesses (S letters))
                      (GameState (S guesses) letters)
```

für einen Buchstaben bei laufenden Spiel entweder:

- falsch geraten -> einen Versuch weniger
- richtig geraten -> einen Buchstabe weniger

```
processGuess : (letter : Char) ->
               GameState (S guesses) (S letters) ->
               Either (GameState guesses (S letters))
                      (GameState (S guesses) letters)
processGuess letter (MkGameState word missing) =
  case isElem letter missing of
    Yes ind => Right (MkGameState word (remove ind missing))
    No _ => Left (MkGameState word missing)
```

Fertige Spiel Funktion

```
game : GameState (S guesses) (S letters) -> IO Finished
game {guesses} {letters} gameState = do
  (_ ** Letter letter) <- readGuess
  case processGuess letter gameState of
    Left nope => do
      putStrLn "falsch geraten"
      case guesses of
        Z => pure (Lost nope)
```

```
        S k => game nope
    Right yeah => do
      putStrLn "richtig geraten"
      case letters of
        Z => pure (Won yeah)
        S k => game yeah
```

Komplett

```
module Main

import Data.Vect

----------------------------------------------------------------------
-- Hilffunktion


remove : (x : a) -> (xs : Vect (S n) a) ->
         { auto prf : Elem x xs } -> Vect n a
remove {prf = Here} x (x :: ys) = ys
remove {prf = (There Here)} x (y :: (x :: xs)) =
    y :: xs
remove {prf = (There (There later))} x (y :: (z :: xs)) =
    y :: remove x (z::xs)


----------------------------------------------------------------------
-- Spiel-Zustand

data GameState : (guessesRemaining : Nat) ->
                 (letters : Nat) ->
                 Type where
  MkGameState : (word : String)
                -> (missing : Vect letters Char)
                -> GameState guessesRemaining letters


data Finished : Type where
  Lost : (game : GameState 0 (S letters)) -> Finished
  Won : (game : GameState (S guesses) 0) -> Finished
```

```
-----------------------------------------------------------------
-- Spielereingabe prüfen

data ValidInput : List Char -> Type where
  Letter : (c : Char) -> ValidInput [c]


toFew : ValidInput [] -> Void
toFew (Letter _) impossible

toMuch : ValidInput (c::c'::cs) -> Void
toMuch (Letter _) impossible

isValidInput : (cs : List Char) -> Dec (ValidInput cs)
isValidInput [] = No toFew
isValidInput [c] = Yes (Letter c)
isValidInput (c::c'::cs) = No toMuch


isValidString : (s : String) -> Dec (ValidInput (unpack s))
isValidString s = isValidInput _

-----------------------------------------------------------------
-- Eingabe

readGuess : IO (c ** ValidInput c)
readGuess = do
  putStrLn "Buchstabe? "
  input <- getLine
  case isValidString (toUpper input) of
    Yes prf => pure (_ ** prf)
    No _ => do
      print "ungültige Eingabe"
      readGuess

-----------------------------------------------------------------
-- Verarbeitung

processGuess : (letter : Char) ->
               GameState (S guesses) (S letters) ->
               Either (GameState guesses (S letters))
                      (GameState (S guesses) letters)
processGuess letter (MkGameState word missing) =
  case isElem letter missing of
    Yes ind => Right (MkGameState word (remove letter missing))
```

```
    No _ => Left (MkGameState word missing)


--------------------------------------------------------------------
-- Spiel-Funktion

game : GameState (S guesses) (S letters) -> IO Finished
game {guesses} {letters} gameState = do
  (_ ** Letter letter) <- readGuess
  case processGuess letter gameState of
    Left nope => do
      putStrLn "falsch geraten"
      case guesses of
        Z => pure (Lost nope)
        S k => game nope
    Right yeah => do
      putStrLn "richtig geraten"
      case letters of
        Z => pure (Won yeah)
        S k => game yeah


--------------------------------------------------------------------
-- Main

main : IO ()
main = do
  ergebnis <- game {guesses=5}
    (MkGameState "DevOpenSpace"
                 ['D','E','V','O','P','N','S','A','C'])
  case ergebnis of
    Lost (MkGameState word _) =>
      putStrLn ("Verloren - das Wort war " ++ word)
    Won _ =>
      putStrLn "Gewonnen!"
```