Nachwort zum Vortrag

Dieser Vortrag richtete sich ausschließlich an erfahrene Programmierer imperativ geprägter Sprachen. Dementsprechend wird vieles ohne Erwähnung vorausgesetzt. Wichtige Konzepte ohne Überraschungen gegenüber anderer Sprachen (z.B. Modul-System) werden, wenn überhaupt, nur am Rande mündlich erwähnt.
Die Folien sind bewusst sehr "dicht" gestaltet. Mein Ziel war es, in kürzester Zeit zu den interessanten fortgeschrittenen Features von Haskell zu gelangen. Die fundamentalen Unterschiede erlauben jedoch kaum einen Verzicht bei den grundlegenden Konzepten. Es gibt fast keine inhaltlichen Wiederholungen; auf nur einmal gezeigte Konzepte wird trotzdem zurückgegriffen. Ich habe versucht, mich bei allen Konzepte auf deren reine Anwendung zu konzentrieren, auf die Theorie dahinter wurde verzichtet.

Was ich aus dem erhaltenen Feedback lernte:

- Ich habe zu oft "Mathematik" erwähnt; Folie 7 größtenteils streichen:

  Was ich vermitteln wollte: Mathematik hat über Jahrhunderte eine präzise Notation geschliffen. Sich an dieser Notation zu orientieren könnte sinnvoller sein, statt das Rad neu zu erfinden (Fkt.def., list comprehension, where-clause, etc.)
  Verstanden wurde aber: Haskell ist gut für rein mathematische Berechnungen.

  Haskell ist kein Ersatz für Matlab oder Maple, sondern eine allgemeine Programmiersprache!

- *Monaden* sind wichtig und schwierig, aber in der Anwendung eigentlich einfach. Das war bekannt, deshalb verzichtete ich auf die monadischen Gesetze. Dennoch: Folie 24 ist viel zu abstrakt, ein konkretes IO Beispiel mit Angabe der Ausgabe/Eingabe anstatt der Typen wäre besser. Das Wort "Monade" hätte ich nicht aussprechen sollen, eine einzelne Erwähnung auf Folie 24 reicht. Auf Folie 25 und 26 hätte ich verzichten sollen. Folie 27 ist ok, jedoch das Wort Monade streichen.

- Anstatt Par Monade hätte ich vielleicht besser STM behandelt, da Begriff auch außerhalb von Haskell von Interesse ist. (Hängt natürlich von den Zuhörern ab.)

- Unbedeutende Details verschweigen, auch wenn es stillschweigend falsche Fakte vermitteln könnte (z.B. Folie 21: Unterschied zwischen Haskell's nicht-strikt Semantik und GHC's lazy evaluation).

- Es wäre schön, die Inhalte anhand eines Vergleichs zwischen einem Haskell und Python Programm größeren Umfangs als running-example zu machen. Doch ich habe wenig Hoffnung, dies zu realisieren, selbst wenn ich ausgiebige Erfahrungen in anderen Programmiersprachen hätte:
  Haskell bietet Sicherheiten und schnelle Entwicklung gegen Einschränkungen. Ein erfahrener Entwickler, der Zeit hat sein Programm gründlich zu überdenken und zu testen, wird in dieser Zeit die Freiheiten einer imperativen Sprache sicher besser ausnutzen können (wenn er weiß welche Konstellation nie auftreten können – doch der formelle Auschluß ist of schwierig).
  Haskell kann sein Stärke meiner Ansicht nach nur ausspielen, wenn das Programm zu groß und komplex wird, als das ein einzelner Entwickler alle Aspekte im Kopf behalten kann (womit es ungeeignet für einen Vortrag wäre); oder wenn es wenig Zeit gibt, das Programm zu optimieren (womit die Kritik käme, so würde das ja niemand programmieren).
  Alternativ mag ein mit dem Yesod Framework erstellter Webserver mit kurzem Code beeindrucken (z.B. Twitter-Clone Tutorial), doch das Programm erfordert viele fortgeschrittene Konzepte (TemplateHaskell, Monaden, etc.) und wäre der Kritik ausgesetzt, dass die ganze "echte" Arbeit ja von dem Framework erledigt wird – und daher nur den Eindruck einer "Baukasten"-Sprache vermittelt.
  Kurze elegante Beispiele, wie Fibonacci auf Folie 22, welche sich nur jeweils auf einen Aspekt konzentrieren, halte ich daher immer noch für den praktikableren Ansatz.

Über weiteres Feedback würde ich mich freuen: Steffen Jost ⟨jost@tcs.ifi.lmu.de⟩

Überblick über Haskell

Steffen Jost

Munich

19. Februar 2013

Haskell ist toll!

Haskell macht es einfacher und billiger große Softwaresysteme zu erstellen und zu warten:

- Schnell kann schneller Code erzeugt werden
  – auch ohne Expertenwissen
- Code ist kompakt, gut lesbar
  – auch leicht zu verifizieren
- Stark typisiert: was kompiliert funktioniert
  – auch ohne langwierige Tests
- Nebenläufigkeit ist unproblematisch

> *I learned Haskell a couple of years ago, having previously programmed in Python and (many) other languages. Recently, I've been using Python for a project (. . .), and find my Python programming style is now heavily influenced (for the better, I hope ;-) by my Haskell programming experience.*
>
> *(Graham Klyne, Haskell-Wiki)*

Haskell's Probleme

- Völlig andere (mathematische) Denkweise
- Großer Kontrollverlust gegenüber herkömmlichen Sprachen
- Stark typisiert: viele Fehlermeldung bis es mal kompiliert
- Keine kommerzielle Unterstützung (IDE, Debugger,. . .)
- Maschinen-nahe Hand-Optimierung sehr schwierig
- Gute "funktionale" Features wurden mittlerweile in anderen Sprachen übernommen GC, HO Fkt., Anonyme Fkt., etc.

Einschränkungen werden oft als sehr unangenehm empfunden – jedoch vorteilhaft für Mehrpersonen-Projekte und Wartung!

Trotzdem erste kommerzielle Erfolge:
Finanzwesen, Telekommunikation (Ericsson mit Erlang)

Haskell

- Effektfreie funktionale Sprache mit verzögerter Auswertung
- Benannt nach Haskell Curry (1900-82), Logiker
- Beliebtes Testobjekt der Informatik
- Standards: Haskell98 und Haskell2010
- Verschiedene Implementierung verfügbar;

  Wichtigste ist die **Haskell Platfrom**:
  - GHC: Glasgow/Glorious Haskell Compiler — Hammond, 1989
  - Compiler, Interpreter und Standard-Bibliotheken
  - Lead-Developers:
    Simon Peyton-Jones — Microsoft Research Cambridge
    Simon Marlow — Facebook

Haskell vs. Python

	Python	Haskell
Primäres Paradigma	imperativ	deklarativ
Speicherzustände	veränderlich	rein funktional
Typisierung	dynamisch	statisch
Auswertung	strikt	faul
Whitespace	sensitiv	sensitiv
Speicherverwaltung	automatisch	automatisch
Jahrgang	1991	1990

Vorsicht vor allen Verallgemeinerungen!

Haskell ist deklarativ

Philosophie deklarativer Sprachen:

Spezifiziere was berechnet werden soll,
nicht wie es berechnet wird

Deklarative Sprachen durch Mathematik motiviert:

Prädikaten-Logik	$\Rightarrow$	Prolog
λ-Kalkül	$\Rightarrow$	Haskell
Relationale Algebra	$\Rightarrow$	SQL

- Rein deklarative Sprachen problematisch in Praxis, denn Ressourcenverbrauch hängt kritisch vom *wie* ab
  (Prolog, SQL ohne join)
- Funktional Programmierung geht einen Mittelweg:
  Berechnung bleibt deterministisch

Haskell ist *rein* funktional

Funktional:

Basis Konzept ist die mathematische Funktion:

```
double x = x + x                    -- Funktion mit 1 Argument
foo x y z = x + y * double z                -- mit 3 Argumenten
add = \x y -> x + y                 -- Anonyme Fkt. 2 Argumente
```

Funktion sind auch ganz normale Werte $\Rightarrow$ Modularisierung

```
twice f x = f x x           -- Funktion mit Funktionsargument
double2 = twice (+)         -- Funktionsanwendung auf Funktion
```

Die Funktion `double2` ist nicht unterscheidbar von `double`.
Funktionsanwendung links-assoziativ: `f x y z == ((f x) y) z`

Referentielle Transparenz:

Wert einer Variablen ist *unveränderlich*!

Andere funktionale Sprachen verzichten auf Reinheit und kennen imperative Befehle z.B. F#, Scala, SML, OCaml, Erlang

Haskell ist *rein* funktional

Funktional:

Basis Konzept ist die mathematische Funktion:

```
double x = x + x                    -- Funktion mit 1 Argument
foo x y z = x + y * double z                -- mit 3 Argumenten
add = \x y -> x + y                 -- Anonyme Fkt. 2 Argumente
```

Funktion sind auch ganz normale Werte $\Rightarrow$ Modularisierung

```
twice f x = f x x           -- Funktion mit Funktionsargument
double2 = twice (+)         -- Funktionsanwendung auf Funktion
```

Die Funktion `double2` ist nicht unterscheidbar von `double`.
Funktionsanwendung links-assoziativ: `f x y z == ((f x) y) z`

Referentielle Transparenz:

Wert einer Variablen ist *unveränderlich*!

Andere funktionale Sprachen verzichten auf Reinheit und kennen imperative Befehle z.B. F#, Scala, SML, OCaml, Erlang

Haskell ist *rein* funktional

Funktional:

Basis Konzept ist die mathematische Funktion:

```
double x = x + x                    -- Funktion mit 1 Argument
foo x y z = x + y * double z                -- mit 3 Argumenten
add = \x y -> x + y                 -- Anonyme Fkt. 2 Argumente
```

Funktion sind auch ganz normale Werte $\Rightarrow$ Modularisierung

```
twice f x = f x x           -- Funktion mit Funktionsargument
double2 = twice (+)         -- Funktionsanwendung auf Funktion
```

Die Funktion `double2` ist nicht unterscheidbar von `double`.
Funktionsanwendung links-assoziativ: `f x y z == ((f x) y) z`

Referentielle Transparenz:

Wert einer Variablen ist *unveränderlich*!

Andere funktionale Sprachen verzichten auf Reinheit und kennen imperative Befehle z.B. F#, Scala, SML, OCaml, Erlang

Referentielle Transparenz bedeutet

- Wert einer Variablen ist *unverändelich*
- Ein Ausdruck wertet immer zum gleichen Wert aus
- Keine Seiteneffekte!
  $\Rightarrow$ Programme sind kompositional, Lokal verstehbar
- Erleichtert Testen und Verifikation $\Rightarrow$ nur Gleichungsrechnen
- Berechnungsreihenfolge unbedeutend
  $\Rightarrow$ Optimierung, Faule Auswertung, Parallelisierbarkeit
- Persistenz & Sharing: Datenstrukturen verändern erzeugt Kopie, doch unveränderte Teile können referenziert werden
  $\Rightarrow$ effiziente funktionale Datenhaltung möglich (Okasaki)

Flusskontrolle in Haskell

Es gibt keine Anweisungen, sondern nur Ausdrücke, d.h.

- es gibt kein `if` ohne `else` Haskell's `if b then x else y` entspricht `b ? x : y` in C/Java und `x if b else y` in Python
- es gibt kein `goto`
- es gibt keine Schleifen*

Es gibt aber Pattern Matching, Fallunterscheidung und Rekursion:

```
bar x y z = case y of
              0                    -> "Zero"
              1                    -> show (x + z)
              y | y < 0            -> "Don't be negative"
                | y < 10 && y > 5  -> "Just right!"
                | otherwise        -> bar x (y-3) z
```

Flusskontrolle in Haskell

Es gibt keine Anweisungen, sondern nur Ausdrücke, d.h.

- es gibt kein `if` ohne `else` Haskell's `if b then x else y` entspricht `b ? x : y` in C/Java und `x if b else y` in Python
- es gibt kein `goto`
- es gibt keine Schleifen*

Es gibt aber Pattern Matching, Fallunterscheidung und Rekursion:

```
bar _ 0 _              = "Zero"
bar x 1 z              = show (x + z)
bar x y z
  | y < 0              = "Don't be negative"
  | y < 10 && y > 5    = "Just right!"
  | otherwise          = bar x (y-3) z
```

Haskell ist stark typisiert

Well-typed programs can't go wrong!

(Milner, 1978)

- Keine Typfehler zur Laufzeit
- Frühe Fehlererkennung
- Schnellere Ausführung
- Auflösung von überladenen Funktionen
- Typen = Dokumentation

Typen werden größtenteils inferiert.

Basistypen: `Char, Int, Integer, Double, Real,...`

Funktionstypen: `Int -> Int, (Int -> Int) -> (Int -> Int),...`

Algebraische Datentypen: Tupel, Enums, Listen, Records,...

Beispiele: Funktionstypen

```
double :: Integer -> Integer
double x = x + x

foo :: Int -> Int -> Int -> Int
foo x y z = x + y * double z

twice :: (Int -> Int -> Int) -> Int -> Int
twice f x = f x x
```

Funktionstypen sind implizit rechts-geklammert

```
foo      :: Int -> (Int -> (Int -> Int))
```

dies erlaubt "partielle" Applikation Stichwort: "currying"

```
foo 42   ::          Int -> (Int -> Int)
foo 42 3 ::                  Int -> Int
```

Beispiele: Algebraische Datentypen

Tupel: (, , ,)

```
addpair :: (Int,Int) -> Int
addpair (0,y) =   y
addpair (x,y) = x+y
```

Listen: [, , ,]

```
reverse :: [Char] -> String  -- reverse [1,2,3] == [3,2,1]
reverse []    = []
reverse (h:t) = reverse t ++ [h]

concat :: [[Int]] -> [Int] -- concat [[1,2],[3,4]] == [1,2,3,4]
```

Spezielle Syntax für Listen:

```
[1,2,3,4]    -- ist Kurzform von 1:2:3:4:[]
['c'..'g']   -- evaluiert zu "cdefg"
[1,3..10]    -- evaluiert zu [1,3,5,7,9]
```

List Comprehension in Anlehnung an ZF Aussonderungsaxiom

```
[ (x,z) | x <- [1,3..5], y <- [0..x], even y, let z = y+1 ]
```

evaluiert zu `[(1,1),(3,1),(3,3),(5,1),(5,3),(5,5)]`

Beispiele: Aufzählungen und Records

Records:

```
data Person  = Mann { name:: String, alter  :: Int    }
             | Frau { name:: String, gewicht:: Double}

p1 = Mann { name = "Kunibert", alter = 77 }
p2 = Frau "Kunigunde" undefined
p3 = p2 { gewicht = 77.7 }          -- p3 nutzt namen von p2
```

Pattern-Matching kann partiell sein:

```
istFrau :: Person -> Bool
istFrau Frau {} = True
istFrau   _     = False
```

Projektionen automatisch definiert:

```
name    :: Person -> String
alter   :: Person -> Int
```

Enums:

```
data Bool = True | False
data Person' = Mann'  String  Int | Frau'  String  Double
data IntList = Nil | Cons Int IntList      -- auch rekursiv!
```

Polymorphie

Erlaubt Wiederverwendung und Verallgemeinerung von Code

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y =  f y x
```

Typkonstuktoren können parametrisiert sein:

```
data Maybe b = Just b | Nothing
data List  a = Cons a (List a) | Nil

head :: [a] -> Maybe a
head []    = Nothing
head (h:_) = Just h

map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

Typsicherheit ist weiterhin gewährleistet!

Typklassen (Interfaces)

```
groupBy :: (a -> a -> Bool) -> [a] -> [[a]]
group   :: Eq a =>             [a] -> [[a]] -- Ueberladen

class  Eq a  where
  (==), (/=) :: a -> a -> Bool

instance Eq Person
  (Mann n1 a1) == (Mann n2 a2) = n1 == n2 && a1 == a2
  (Frau n1 g1) == (Frau n2 g2) = n1 == n2 && g1 == g2
  _p1          == _p2          = False
```

Einfacher ist hier die automatische Instanzherleitung:

```
data Person' = Mann'  String  Int | Frau'  String  Double
  deriving (Eq, Show)
```

Abgeleitete Instanzen möglich:

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

Haskell's Antwort zu Overloading, Dynamic Dispatch, Duck-Typing

Syntactic Sugar: Optionen zur Lesbarkeit

```
printPercent :: Double -> String  -- Prozentzahl drucken
printPercent x =
  let rx = (fromIntegral (round (1000.0*x))) / 10.0
      lz = if rx < 10.0 then "0" else ""
  in lz ++ (show rx) ++ "%"

printPercent x  = lz ++ (show rx) ++ "%" {- Prozent drucken -}
  where
    lz = if rx < 10.0 then "0" else ""
    rx = (fromIntegral $ round $ 1000.0*x) / 10.0
```

Lokale Definitionen mit spezieller `where` Syntax hinten anstellbar;
für Top-Level Definitionen und Case, auch für Pattern-Guards.

`$` ist rechts-assoziativer Infix Operator für Funktionsanwendung:

```
($)   :: (a -> b) -> a -> b
f $ x =  f x
```

macht nichts, aber niedrige Bindung spart Klammern!
Spezialfall: `map ($0) :: Num a =>[a -> b] -> [b]`

Haskell ist faul

Haskell: nicht-strikte Semantik; GHC: implementiert lazy eval.
D.h.: es wird nur ausgewertet, was benötigt wird; und nur einmal

- Beschleunigt die Ausführung erheblich

  ```
  map double $ map double $ map double [1..10]
  ```

  iteriert nur *einmal* über die Liste
- Ignoriert Fehler in unbenötigten Code

  ```
  > let x = map (10/) [10,5,2,0,undefined,4,1]
  > x!!5
  2.5
  > x
  [1.0,2.0,5.0,Infinity,*** Exception: Prelude.undefined
  ```

- Erlaubt Rechnen mit "unendlichen" Datenstrukturen

  ```
  > take 10 [1..]
  [1,2,3,4,5,6,7,8,9,10]
  > take 10 $ [1..3] ++ x
  [1,2,3,1,2,3,1,2,3,1]
  ```

Haskell ist faul

- Verzögerte Auswertung erlaubt zirkuläre Programme:

```
fibs :: [ Integer ] -- Liste aller Fibonacci Zahlen
fibs = 1 : 1 : (zipWith (+) fibs (tail fibs ))
    {- zipWith :: (a -> b -> c) -> [a] -> [b] -> [c] -}

fib :: Int -> Integer
fib n = fibs !! n     -- memoisation already included!
```

- Trennung von Daten und Kontrollfluss:
  - *Wie* berechnet man Fibonacci Zahlen festgelegt.
  - *Wieviele* benötigt werden wird woanders entschieden.
- Jede Fibonacci Zahl wird nur einmal berechnet!
- Problem: kann zu hohem Speicherverbrauch führen
- Problem: Verhindert Verständnis der Ausführung
  Hauptkritikpunkt an Haskell Ich arbeite dran! ;-)

Diese Punkte beziehen nicht nur auf zirkuläre Programme!

Seiteneffekte in Haskell

I/O besteht nur aus Seiteneffekten. Dazu besitzt Haskell ein imperatives Fragment, welches mit `do` eingeleitet wird:

```
module GoodWorld where
  import System.IO
  import System.Time (getClockTime, ClockTime(..))

  main :: IO ()
  main = do
    (TOD seconds n) <- System.Time.getClockTime
    let greeting = goodX seconds
    System.IO.putStrLn $ greeting  ++ " world!"

  goodX :: Integer -> String
  goodX s =
    | isMorning = "Good morning"
    | otherwise = "Good day"
    where
      isMorning = 12 > s `mod` (60 * 60 * 24)
```

Innerhalb dieses Fragmentes gibt es Schleifen, if-ohne-else, usw.

DO Notation

Hintereinanderausführung mehrerer Befehle mit Effekten:

```
do
  me1                       -- me1 :: M a
  me2                       -- me2 :: M b
  x <- me3                  -- me3 :: M c,  x :: c
  let y = f x               -- f :: c -> d, y :: d
  when b $ me4 y            -- b :: Bool, me4 :: d -> M ()
  z <- forM [1..3] me5      -- me5 :: Int -> M e, z :: [e]
  return z                  -- :: M [e]
```

- Ausführungsreihenfolge wird immer eingehalten
- Effekte werden entsprechend weitergereicht

Stichworte: Monaden und Funktoren

DO Notation

DO-Notation nur Zucker für gewöhnliche Funktionen:

```
do
  me1
  x <- me2
  me3 $ f x
```

kann man auch schreiben als

```
me1 >> me2 >>= \x -> me3 (f x)
```

Gewöhnliche binäre Operatoren definiert in Monaden-Klasse:

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

instance Monad IO
instance Monad Maybe
instance Monad []
```

$\Rightarrow$ rein-funktionaler Code! Keine Spracherweiterung!

DO Notation

DO-Notation nur Zucker für gewöhnliche Funktionen:

```
do
  me1
  x <- me2
  me3 $ f x
```

kann man auch schreiben als

```
me1 >> me2 >>= \x -> me3 (f x)
```

Gewöhnliche binäre Operatoren definiert in Monaden-Klasse:

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

instance Monad IO
instance Monad Maybe
instance Monad []
```

$\Rightarrow$ rein-funktionaler Code! Keine Spracherweiterung!

Beispiel: Zustands Monade

Weiteres Beispiel für Monaden: veränderbare Zustände

```
import Data.STRef.Lazy
import Control.Monad.ST.Lazy

state :: [a] -> [a]
state l = runST $ do    -- runST :: (forall s. ST s a) -> a
  ref_l <- newSTRef l
  l1 <- readSTRef ref_l
  let l1' = reverse l1
  writeSTRef ref_l l1'
  state2 ref_l

state2 :: STRef s b -> ST s b
state2 ref_l = do         -- superfluous "do" and "return"
  l2 <- readSTRef ref_l
  return l2
```

- Code der Zustand liest oder verändern lebt in Monade
- Typesystem verhindert Verwechselung von Referenzen

Haskell ist Nebenläufig

Auswertereihenfolge eh egal ⇒ keine Race Conditions,
also können wir sehr leicht parallel auswerten:

Glasgow parallel Haskell (GpH)

Einfach zwei Primitive einstreuen, fertig:

```
par  :: a -> b -> b  -- Argumente parallel auswerten
pseq :: a -> b -> b  -- Auswertung erzwingen
```

Software Transactional Memory (STM) Nebenläufigkeit zu Fuß:

Explizites forking in IO-Monade, Kommunikation über Variablen mit Semaphoren, Backtracking eingebaut

Par Monade

Die moderne sichere Methode mit Monade

Nested Data Parallelism

Ein paar ausgewählte parallele Operation für große Datenstrukturen, wie z.B. Arrays nutzt Grafik (CUDA)

Par Monade

```
do v1 <- new
   v2 <- new
   fork $ put v1 (f x)
   fork $ put v2 (g x)
   get v1
   get v2
   return (v1 + v2)
```

- Forking ist explizit
- Ergebnisse werden mit write-once IVar's kommuniziert
- Parallele Berechnung sind deterministisch

```
runPar :: Par a -> a        -- teuer
fork :: Par () -> Par ()    -- billig

new :: Par (IVar a)
get :: IVar a -> Par a
put :: NFData a => IVar a -> a -> Par ()
```

Partitionierung verbleibt Aufgabe des Programmierers

Template Haskell

Metaprogrammierung: Haskell Code verarbeitet Haskell Code:

```
ghci -XTemplateHaskell
:m + Language.Haskell.TH

> let x = [| \x -> x+1 |]
x :: Q Exp

> runQ x
LamE [VarP x_0] (InfixE (Just (VarE x_0)) (VarE GHC.Num.+) (Just

> $(x) 3
4
```

Quasi-Quoting: Code in Oxford-Klammer `[| |]` wird zu Daten
Splicing: Daten welche Code darstellen werden mit `$( )` zu Code

- Ebenen könnnen verschachtelt werden!
- Q Monade kümmert sich z.B. um frische Bezeichner

Template Haskell

```
{-# LANGUAGE TemplateHaskell -#}
import Language.Haskell.TH

projNI :: Int -> Int -> ExpQ -- generic projection I-th element
projNI n i = lamE [pat] rhs
  where pat = tupP (map varP xs)
        rhs = varE (xs !! (i - 1))
        xs  = [ mkName $ "x" ++ show j | j <- [1..n] ]

$(projNI 4 3) ('a','b','c','d') -- evaluiert zu 'c'
```

Code von `projNI` wird während des Kompilierens ausgeführt

Template Haskell verarbeitet auch andere Sprachen

```
...
let jSkillCases t = mconcat [jShowiSkill i t | i <- heroTypeList ]
let widget = do
    toWidget [lucius|                                                    -- CSS
      .itemSelector {
        width:  11em;
        height: 13em;
      }
    |]
    addScriptRemote "http://ajax.googleapis.com/ajax/libs/jquery/1.8.2/jquery.min.js"
    toWidget [julius|                                                    -- JavaScript
      function Aendern () {
        var tc = document.getElementById('#{fvId htypeView}');
        switch (tc.value) {
          ^{jSkillCases}
        }
      }
      document.getElementById('#{fvId htypeView}').setAttribute('onchange','Aendern()');
    |]
    forM_ heroTypeList (\i -> toWidget [julius|                          -- JavaScript
        function Show#{show i}Skill () {
          document.getElementById('heroImage').setAttribute('class','#{displayHero i}');
          document.getElementById('SkillAtag').innerHTML = '#{displayHeroSkill i 1}';
        }
    |])
    [whamlet|                                                            --HTML
      <table>
        <tbody align="right">
          <tr>
            <td>
              #{fvLabel aliasView} #
              ^{fvInput aliasView}
            <td>
...
```

Yesod Framework

Haskell ist . . .

- deklarativ
- rein funktional
- sehr modular
- statisch getypt
- faul ausgewertet
- leicht nebenläufig
- fähig andere Sprachen einzubetten
- whitespace sensitiv

. . . einfach toll!

Weiterführende Informationen und Quellen:

Haskell Platform	haskell.org	
Bibliothek	www.haskell.org/ghc/docs/latest/html/libraries	

Haskel Tutorials:

Learn You a Haskell for Great Good	learnyouahaskell.com	2010
Real World Haskell	book.realworldhaskell.org/read	2008
A Gentle Introduction To Haskell	www.haskell.org/tutorial	2000

(Alle Links sind anklickbar.)