

# FUNKTIONALE PROGRAMMIERUNG

## GRUNDLAGEN DER FUNKTIONALEN PROGRAMMIERUNG

Andreas Abel und Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,  
Ludwig-Maximilians Universität, München

16. April 2012

# ADMINISTRATIVES

- Vorlesung: Mo 16-18 Uhr, Oettingenstr. 67, U127
- Vorlesung: Di 12-14 Uhr, Oettingenstr. 67, C003
- Übungen: Do 16-18 Uhr, Oettingenstr. 67, Takla-Makan (L11 im CIP-Pool)
- Erste Übung 19.04.: Rechnerübung (GHC, GHCi, cabal)
- Prüfungsleistung: Mini-Projekt mit Notenbonus durch Übungspunkte
- Bitte anmelden auf Uniworx!

- Wiederholung: Grundlagen funktionaler Programmiersprachen: Rekursion, Listen, Typen, Funktionen höherer Ordnung, benutzerdefinierte Datentypen und Musterabgleich; Auswertungsstrategie
- Fortgeschrittene Konzepte: Überladung durch Typklassen, (Seiten-)effekte durch Monaden, Wertrekursion und unendliche Strukturen
- “Real-world programming” : Grafische Benutzeroberflächen, Webapplikationen, Parallelität und Nebenläufigkeit
- Qualitätssicherung: Dokumentation, systematisches Testen, Profiling
- Forschungsthemen im Design von Programmiersprachen: Abhängige Typen, Verifikation

Erläutert anhand der Programmiersprachen Haskell und Agda.

# GRAU IST ALLE THEORIE

Voraussetzungen:

- Vorlesung: Programmierung und Modellierung
- Grundkenntnisse zur Programmierung in einer funktionalen Sprache (z.B. SML)

Praxis:

- Wöchentliche Programmier-Übungen
- Wir verwenden den Compiler GHC
- Kleines Abschlussprojekt (Graphische oder Web-Applikation)



## Philosophie *deklarativer Sprachen*

*Spezifiziere **was** berechnet werden soll, **nicht wie** es berechnet werden soll.*

Imperative Sprachen starten von low-level Code (z.B. Assembler) und führen Abstraktionen ein um die Programmierung zu vereinfachen (z.B. Schleifen).

Deklarative Sprachen sind durch die Sprache der Mathematik motiviert und beschreiben Funktionen oder Beziehungen:

- der  $\lambda$ -Kalkül beschreibt Funktionen und ist die Basis für **funktionale Sprachen**
- die Prädikaten-Logik beschreibt Beziehungen und ist die Basis für **logische Sprachen**

# DEKLARATIVES PRINZIP: HARTE LANDUNG

- Das reine deklarative Prinzip scheitert an der Praxis:  
Der Ressourcenverbrauch hängt kritisch vom *wie* der Berechnung ab.
- Der Fall PROLOG:
  - Prinzip: Prädikatenlogik.
  - Nichtdeterminismus: Lösungen der Problemspezifikation werden (blind) *gesucht*.
  - Das *fifth generation computing project* (Japan 1980er) scheiterte.
  - PROLOG benötigt extralogische Konzepte (z.B. *cuts*).
- Erfolgreich: SQL
  - Prinzip: Relationale Algebra.
  - Rein deklarative Anfragen (*cross join* und *where*-Klauseln) zu ineffizient.
  - Explizite *join*-Klauseln effizienter.

# IN DER MITTE: FUNKTIONALE PROGRAMMIERUNG

- Funktionale Programmierung hat deklarativen Charakter. (Erreicht durch hohen Abstraktionsgrad.)
- Deterministisch: Das *wie* der Berechnung ist jedoch festgelegt (keine implizite Suche).
- Funktionale Programme können in Maschinensprache übersetzt werden.
- Einbindung von fremden Bibliotheken mittels *foreign function interface*.



# PRINZIP 1: REFERENTIELLE TRANSPARENZ

- Der Wert einer Variablen wird zu Beginn zugewiesen und ist dann *unveränderlich*.

```
fun bla y =      (* value of y is fixed *)  
  let val x = y * y (* value of x is fixed to y * y *)  
    val x = x + 1 (* new variable with same name! *)  
  in x + y  
end
```

- *Referentielle Transparenz*: Der Wert eines Ausdruck hängt nur von den (konstanten!) Werten der Variablen ab.
- Wird ein Ausdruck zweimal ausgewertet, kommt der selbe Wert heraus.
- Die Bedeutung einer Funktion ist aus ihrem Kode allein ermittelbar!
- Programme sind kompositional (verstehbar).

# VORTEILE REFERENTIELLER TRANSPARENZ

- Berechnungen parallelisierbar.
- Ermöglicht Optimierungen durch den Compiler, z.B. *common subexpression elimination*.  
 $(x + y) * (x + y)$  optimiert zu  
**let val**  $z = x + y$  **in**  $z * z$
- Seiteneffekte wie Wert-Veränderung verhindern solche Optimierungen:  
 $(++x + y) * (++x + y)$
- Erleichtert Testen und Verifikation:  
Imperative Sprachen benötigen Hoare-Logik, Haskell nur Gleichungsrechnen.  
(Alter Name: GOFER = Good For Equational Reasoning.)
- Unveränderliche Variablen gibt es auch in C (Schlüsselwort `const`), aber es ist nicht der *default*.

## PRINZIP 2: UNVERÄNDERLICHE DATENSTRUKTUREN

- Datenstrukturen sind auch unveränderlich: Einfügen in eine Liste erzeugt eine *neue* Liste.
- *Persistenz*: Die alte Liste existiert auch noch.
- *Sharing*: Unveränderte Restliste wird von alter und neuer Liste gemeinsam referenziert.
- *Garbage Collection*: Nicht mehr benötigte alte Versionen werden aus dem Speicher entfernt.

# VORTEILE UNVERÄNDERLICHE DATENSTRUKTUREN

- Typische Falle veränderlicher Strukturen: Veränderung während Iteration (`for`).
- Backtracking (Rekursion): Alter Zustand ist noch vorhanden, muss nicht wiederhergestellt werden.
- Nebenläufigkeit: Kein explizites Kopieren der Struktur nötig, wenn 2 Threads manipulieren wollen.

## PRINZIP 3: ABSTRAKTION

- Extrahieren von gemeinsamen Codefragmenten leicht gemacht durch *Funktionen höherer Ordnung*.

```
fact n = foldl (*) 1 [1..n]
```

- Entwicklung von universell verwendbaren Bausteinen. (Besser als Designpatterns.)
- Wiederverwendbarkeit durch *Polymorphismus*.

# PRINZIP 4: STARKE TYPISIERUNG

- Typsystem dient:
  - der Auflösung von Überladung,
  - der Aufdeckung von Fehlern durch den Übersetzer,
  - der Dokumentation von Funktionen,
  - der Suche passender Funktionen.
- Typen werden inferiert (ML-Dialekte), oder zumindest zum größten Teil (Haskell, Scala).
- Fortgeschrittene Typsysteme dienen der Verifikation (Agda).

- Effektfreie funktionale Sprache mit verzögerter Auswertung.
- Benannt nach *Haskell Curry* (1900-82).
- Standards: **Haskell 98** und **Haskell 2010**.
- Implementierungen: Hugs, NHC, ..., **GHC**.
- **Glorious Haskell Compiler** entwickelt zuerst an der Uni Glasgow und nun bei Microsoft Research Cambridge.
- Die Hauptentwickler Simon Peyton-Jones und Simon Marlow bekamen 2011 den ACM SIGPLAN *Programming Language Software Award*.
- Installation: **Haskell Platform** herunterladen.  
Weitere Bibliotheken: `cabal install foo`

# HASKELL-SYNTAX

- Wenig Schlüsselwörter.
- Wenig Klammerung.
- **Einrückung** ist Teil der Syntax! (wie by Python)
- Keine Wertzuweisung ( $:=$ ).
- Benutzerdefinierte **Infixoperatoren**.
- Benutzerdefinierte **überladene Operationen** mittels **Typklassen**.
- Simplex hierarchisches Modulsystem; Modulnamen entsprechen Dateinamen.



# SML vs. HASKELL-SYNTAX: FUNKTIONEN

## SML-Syntax

```
(* SML comment block *)
```

```
(* function abstraction *)
```

```
fn x => fn y => x
```

```
fn (x,y) => y
```

```
fn [] => true  
| (x::xs) => false
```

```
(* function declaration *)
```

```
fun iter f a 0 = a  
| iter f a n =  
    iter f (f a) (n-1)
```

## Haskell-Syntax

```
{- Haskell comment block -}  
-- Haskell one line comment
```

```
-- \ for λ
```

```
\ x y -> x
```

```
\ (x,y) -> y
```

```
{- no multi-clause anonymous  
    functions -}
```

```
-- just equations
```

```
iter f a 0 = a
```

```
iter f a n =
```

```
    iter f (f a) (n-1)
```

# SML vs. HASKELL-SYNTAX: LISTEN

Rollentausch: In Haskell ist `:` Listenkonstruktion (SML `::`) und `::` Typzuweisung (SML `:`).

<code>[]</code>	<i>(* empty list *)</i>	<code>[]</code>
<code>x::xs</code>	<i>(* cons *)</i>	<code>x:xs</code>
<code>[1,2,3]</code>	<i>(* literal *)</i>	<code>[1,2,3] {- or -} [1..3]</code>
<code>l @ l'</code>	<i>(* append *)</i>	<code>l ++ l'</code>

**Typsignaturen** bezeichner `::` typ sind optional.

```
fun map f [] = []
  | map f (x :: xs) =
    f x :: map f xs
```

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

# SML vs. HASKELL-SYNTAX: DATENTYPEN

Datenkonstruktoren sind in Haskell **gecurryte** Funktionen.

```
datatype 'a tree
  = Leaf of 'a
  | Node of 'a tree * 'a tree

(* Leaf : 'a -> 'a tree *)
fun node l r = Node (l, r)

(* case distinction *)
case t
  of Leaf(a)   => [a]
     | Node(l,r) => f l @ f r

data Tree a
  = Leaf a
  | Node (Tree a) (Tree a)

-- Leaf :: a -> Tree a
{- Node :: Tree a -> Tree a ->
   Tree a -}

-- similar to SML
case t of
  Leaf a   -> [a]
  Node l r -> f l ++ f r
```

Datentypen und Konstruktoren müssen in Haskell **Groß** geschrieben werden. In SML hat Großschreibung keine lexikalische Signifikanz.

# SML vs. HASKELL-SYNTAX: LOKALE DEFINITIONEN

In Haskell können Definitionen mit **where** nachgestellt werden.

```
let val k    = 5
     fun f 0  = k
       | f n
= f (n-1) * n
in  f k
end
```

*(\* no post-definition \*)*

```
let k    = 5
     f 0  = k
     f n  = f (n-1) * n
in  f k

f k where
     f 0  = k
     f n  = f (n-1) * n
     k    = 5
```

**local ... in ... end**                    *-- nothing corresponding*

SML bearbeitet Definitionsfolgen von vorne nach hinten. In Haskell ist die Reihenfolge der Definitionen unerheblich.

# SML vs. HASKELL-SYNTAX: REKURSION

In Haskell sind alle Gleichungen wechselseitig rekursiv.

```
(* mutual recursion *)
fun even 0 = true
  | even n = odd (n-1)
and odd 0 = false
  | odd n = even (n-1)

val rec f = fn x =>
  if x <= 1 then 1
  else x * f (x-1)

-- no need to mark mutual rec.
even 0 = true
even n = odd (n-1)
odd 0 = false
odd n = even (n-1)

f = \ x ->
  if x <= 1 then 1
  else x * f (x-1)
```

In SML benötigt Rekursion das Schlüsselwort **rec** oder **fun...and....**

# SML vs. HASKELL-SYNTAX: MODULE

Haskell hat ein simples Modulsystem (keine Signaturen, keine Funktoren).

```
structure M = struct  
  ...  
end
```

```
open Text.Pretty  
structure S = System.IO
```

```
module M where  
  ...
```

```
import Text.Pretty  
import qualified System.IO as S
```

# WÄCHTER IN FALLUNTERSCHIEDUNGEN

- In Fällen in Funktionsdefinition oder **case** können durch **Wächter** (engl. *guards*) “geschützt” werden.
- Z.B. Definition von **filter** p l, das alle Elemente aus Liste l zurückgibt, die Prädikat p erfüllen.

```
filter :: (a -> Bool) -> [a] -> [a]
```

```
filter p [] = []
```

```
filter p (a:as) | p a = a : filter p as  
                | otherwise = filter p as
```

- Nach den Mustern (wie (a:as)) können nach dem Strich | noch Wächter, Bool'sche Ausdrücke stehen.
- Nur wenn diese zu **True** auswerten, wird der Zweig gewählt, ansonsten zum Nächsten gegangen.
- Wächter **otherwise** ist nur ein Alias für **True**, also immer passierbar.

# ERWEITERTE WÄCHTER

- Neben Bool'schen Wächter gibt es noch die **pattern guards**.
- Folgende Funktion `allJust` nimmt eine Liste von optionalen Werten **Maybe** `a` und prüft, ob alle Elemente definiert sind (**Just** `a`).

- 1 Ist dies der Fall, wird eine Liste der Elemente ohne **Just**-Konstruktor zurückgegeben.
- 2 Andernfalls **Nothing**.

```
data Maybe a = Just a | Nothing
```

```
allJust :: [Maybe a] -> Maybe [a]
```

```
allJust [] = Just []
```

```
allJust (Just a : l) | Just l' <- allJust l = Just (a : l')
```

```
allJust _ = Nothing
```

- Der Wächter `Just l' <- allJust l` prüft, ob das Ergebnis der Rekursion ein **Just** ist und bindet den Inhalt an Variable `l'`.



# NOCHMAL pattern guards

- Berechnet eine Funktion ein Tupel, können wir das Ergebnis des rekursiven Aufrufs mit einem *pattern guard* zerlegen.
- Folgende Funktion **span** p l teilt die Liste l an der Stelle, wo das Prädikat p zum ersten Mal *nicht mehr* gilt.

```
span :: (a -> Bool) -> [a] -> ([a], [a])  
span p [] = ([], [])  
span p (x : xs)  
  | p x, (ys, zs) <- span p xs = (x : ys, zs)  
  | otherwise = ([], x : xs)
```

- Hier findet sich ein Bool'scher Wächter p x und ein *pattern guard* (ys, zs) <- **span** p xs.

# LISTENKOMPREHENSION

- In der Mathematik verwendet man **Mengenkomprehension** wie  $\{x \mid x \in \mathbb{N}, x \bmod 3 = 1\}$ , die Menge der natürlichen Zahlen, die bei Division durch 3 den Rest 1 haben.
- In Haskell gibt es eine analoge Notation für Listen.

```
filter :: (a -> Bool) -> [a] -> [a]
```

```
filter p l = [ a | a <- l, p a ]
```

- Damit ist auch das kartesische Produkt zweier Listen direkt definierbar:

```
cartesian :: [a] -> [b] -> [(a,b)]
```

```
cartesian as bs = [ (a,b) | a <- as, b <- bs ]
```

- Test cartesian [1,2] [3,4,5] ergibt:  
[(1,3),(1,4),(1,5),(2,3),(2,4),(2,5)]

# INFIX-, PRÄFIX- UND SEKTIONSSCHREIBWEISE

- Einen **Infixoperator** wie + kann man in Klammern (x) **präfix** verwenden. Beispiel: (+) 5 3 statt 5 + 3.
- Dadurch ist partielle Applikation möglich:  
`map ((+) 3) [1..5]` (ergibt [4..8]).
- Alternativ dazu sind **Sektionen**: Durch Weglassen eines Operatorargumentes erhält man eine einstellige Funktion.

```
map (3 +) [1,2] == [4,5]
```

```
map (3 -) [1,2] == [2,1]
```

```
map (/ 2) [1,2] == [0.5,1.0]
```

- Jeden Bezeichner kann man in 'backquotes' infix verwenden.

```
if 3 'elem' [1..5] then 1 else 0
```

# INFIXOPERATOREN

- Eigene Infixoperatoren muss man unter Angabe der **Bindungsstärke** deklarieren. Beispiel: Linksassoziatives Exklusiv-Oder:

```
infixl 3 /+/  
(/+/)    :: Bool -> Bool -> Bool  
x /+ y   = not (x == y)
```

- Funktionskomposition (in SML: o) hat die höchste Bindungsstärke:

```
infixr 9 .  
(.)      :: (b -> c) -> (a -> b) -> (a -> c)  
(f . g)  = \ x -> f (g x)
```

- Applikation mit \$ hat die niedrigste Bindungsstärke:

```
infixr 0 $  
($)      :: (a -> b) -> a -> b  
f $ x    = f x
```

# KLAMMEN SPAREN MIT \$

- Wozu noch einen Operator \$ für Applikation? Klammern sparen!
- Das letzte Argument einer Funktion kann man mit \$ abteilen und braucht es dann nicht zu klammern.
- Statt `fun1 (fun2 arg2 (fun3 (fun4 arg4)))`

`fun1 $ fun2 arg2 $ fun3 $ fun4 arg4`

- Weitere Infixoperatoren:

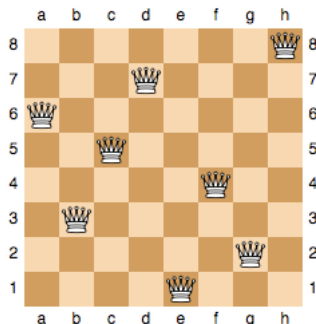
```
infix 4 ==, /=, <, <=, >=, >  -- comparison
infixr 3 &&                    -- boolean and
infixr 2 ||                    -- boolean or
```

# MUSTERBEISPIEL DEKLARATIVER PROGRAMMIERUNG: DIE ACHT DAMEN

Plaziere 8 Damen auf einem Schachbrett, so dass keine eine andere bedroht.

- nur eine pro Spalte
- Lösung ist 8-elementige Liste von Zeilenpositionen.  
Hier: [6, 3, 5, 7, 1, 4, 2, 8]

Statt  $\binom{64}{8} \approx 4$  Milliarden brauchen wir nur  $8^8 \approx 16$  Millionen Damenplazierungen betrachten.



# ZU DEKLARATIV!

```
queens :: Int -> [[Int]]
queens n = [ l
  | l <- lists n [1..n] -- n columns with row positions in 1..n
  , sort l == [1..n]    -- permutation of 1..n (horiz. threats)
  , noDiagThreats l
  ]

-- | Enumerate all lists of length @n@ with elements in @l@.
lists n l | n > 0      = [ x:xs | x <- l, xs <- lists (n-1) l ]
          | otherwise = [[]]

noDiagThreats []      = True
noDiagThreats (x:ys) =
  all (\ (y,n) -> abs(x-y) /= n) (zip ys [1..])
  && noDiagThreats ys
```

# ZU DEKLARATIV!

- Verwendete Listenfunktionen:

```
sort :: Ord a => [a] -> [a]
```

```
zip  :: [a] -> [b] -> [(a,b)]
```

```
all  :: (a -> Bool) -> [a] -> Bool
```

- Program zu langsam (11sec)!
- Pro Zeile darf auch nur eine Dame plziert werden!
- queens erzeugt viele Kandidaten, die keine Permutationen von  $1..n$  sind, die dann wieder aussortiert werden müssen.
- Besser: betrachte nur die  $8! = 40320$  Permutationen von  $1..8$ .



# BETRACHTE NUR PERMUTATIONEN

```
import Data.List (delete)
```

```
queens n = [ l  
  | l <- choose n [1..n] -- permutation of [1..n]  
  , noDiagThreats l  
  ]
```

```
-- | Choose @@ distinct elements from list @l@.
```

```
choose n l  
  | n > 0      = [ x:xs | x <- l  
                      , xs <- choose (n-1) (delete x l) ]  
  | otherwise = [[]]
```

Benötigt nur noch 40msec!

# BETRACHTE NUR GÜLTIGE POSITIONEN

- Wenn Dame plaziert wird, entferne *alle, auch die diagonal* bedrohten Positionen vom Brett.
- Eliminiert frühzeitig weitere Kandidaten.
- choose benötigt nun ein Brett mit noch verfügbaren Positionen.
- Weitere Listenfunktionen:

**zipWith** :: (a -> b -> c) -> [a] -> [b] -> [c]

**filter** :: (a -> Bool) -> [a] -> [a]

# BETRACHTE NUR GÜLTIGE POSITIONEN

```
-- | Produce all queen positionings on a board of size @n*n@.  
queens n = choose (board n)  
  
-- | Initially, for each row, all columns are possible.  
board :: [[Int]]  
board n = take n (repeat [1..n])  
  
choose []      = [[]]  
choose (r:rs) = [ c:cs | c <- r -- pick an available column  
                    , cs <- choose (removeThreatened c rs) ]  
  
-- | Delete column positions in remaining rows @rs@  
--   threatened by a queen in column @c@ of the current row.  
removeThreatened c rs = zipWith (del c) [1..] rs  
  where del c n = filter (\ y -> let d = abs(c-y)  
                          in d /= 0 && d /= n)
```

Benötigt nur noch 4msec!



# ZUSAMMENFASSUNG

- Bisherige Nische für funktionale Sprachen: Compiler, Softwareanalyse und -verifikation, akademische Landschaft.
- Funktionale Programmierung erlebt eine Renaissance:
  - Closures in C#, C++, Java.
  - Funktionale OO-Sprache Scala.
  - Unveränderliche Datenstrukturen für Nebenläufigkeit.
  - Ocaml/F# und Haskell erobern Nische in Finanzbranche.
- Effektfreiheit ermöglicht Optimierung durch Compiler und Parallelisierung.
- Nachteil: Werkzeugkette meist noch nicht komplett oder ausgereift (Bibliotheken, IDE, Debugger, Monitoring...).