

Kapitel 1: Einführung und Grundlagen

- *Teil 1: Einführung und Motivation*
 - Funktionale Programmierung: Warum? Warum mit Haskell?
 - Erste Schritte in Haskell, erste Schritte mit Hugs
- *Teil 2: Grundlagen*
 - Elementare Datentypen
 - Tupel, Listen und Funktionen

Beachte: ...einige Begriffe werden heute im Vorgriff angerissen und erst im Lauf der Vorlesung genau geklärt!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 1

Teil 1: Einführung und Motivation

- Funktionale Programmierung: Warum überhaupt? Warum mit Haskell?
- Erste Schritte in Haskell, erste Schritte mit Hugs

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 2

Warum funktionale Programmierung?

Ein bunter Strauß an *Programmierparadigmen*, z.B.

- *Imperativ*
 - prozedural (Pascal, Modula, C,...)
 - objektorientiert (Smalltalk, Oberon, C++, Java,...)
- *deklarativ*
 - funktional (Lisp, ML, Miranda, Haskell, Gofer,...)
 - logisch (Prolog und Varianten)
- Mischformen
 - z.B. funktional/logisch, funktional/objektorientiert,....
- *Visuell*
 - Stichwort:* Visual Programming Languages (VPLs), z.B. Forms/3, FAR,....
 - Einstieg für mehr: web.engr.oregonstate.edu/~burnett
- ...

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 3

Ein Vergleich – prozedural vs. funktional

Gegeben eine Aufgabe *A*.

Prozedural: Typischer Lösungsablauf in folgenden Schritten:

1. Beschreibe eine(n) Lösung(sweg) *L* für *A*.
2. Gebe *L* in die Form einer Menge von Anweisungen (Kommandos) für den Rechner *unter expliziter Organisation der Speicherverwaltung*.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 4

Zur Illustration ein einfaches Beispiel

Betrachte folgende Aufgabe:

"...bestimme die Werte aller Komponenten eines ganzzahligen Feldes, deren Werte kleiner oder gleich 10 sind."

Eine typische Lösung *prozedural*...

```
...
j := 1;
FOR i:=1 TO maxlength D0
  IF a[i] <= 10 THEN b[j] := a[i]; j := j+1 FI
D0;
```

Mögliches Problem bei großen Anwendungen:

...inadäquates Abstraktionsniveau \leadsto *Softwarekrise!*

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 5

Softwarekrise

- Ähnlich wie objektorientierte Programmierung verspricht deklarative, insbesondere funktionale Programmierung ein angemesseneres Abstraktionsniveau zur Problemlösung zur Verfügung zu stellen
- ...und damit einen Beitrag zur Überwindung der vielzitierten Softwarekrise zu leisten

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 6

Zum Vergleich...

...eine typische Lösung *funktional*, hier in Haskell:

```
...
a :: [Int]
b :: [Int]
b = [ n | n <- a, n <= 10 ]
```

Vergleiche diese funktionale Lösung mit: $\{n \mid n \in a \wedge n \leq 10\}$
...und der Idee, etwas von der Eleganz der Mathematik in die Programmierung zu bringen!

Essenz funktionaler Programmierung

...statt des "wie" das "was" in den Vordergrund stellen!

Hilfsmittel im obigen Beispiel:

- *Listenkomprehension* (engl. list comprehension)
...typisch und spezifisch für funktionale Sprachen!

Noch nicht überzeugt?

Betrachte *Quicksort*, ein komplexeres Beispiel...

Aufgabe: Sortiere eine Liste *L* ganzer Zahlen aufsteigend.

Lösung (mittels Quicksort):

- *Teile:* Wähle ein Element *l* aus *L* und partitioniere *L* in zwei (möglicherweise leere) Teillisten *L*₁ und *L*₂ so, dass alle Elemente von *L*₁ (*L*₂) kleiner oder gleich (größer) dem Element *l* sind.
- *Herrsche:* Sortiere *L*₁ und *L*₂ mit Hilfe rekursiver Aufrufe von Quicksort.
- *Zusammenführen der Teilergebnisse:* Trivial (die Gesamtliste entsteht durch Konkatenation der sortierten Teillisten).

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 8

Quicksort...

... eine typische *prozedurale* (Pseudocode-) Realisierung:

```
quicksort (L, low, high)
  if low < high
    then splitInd = partition(L, low, high)
       quicksort(L, low, splitInd-1)
       quicksort(L, splitInd+1, high)  fi
partition (L, low, high)
  l = L[low]
  left = low
  for i=low+1 to high do
    if L[i] <= l then left = left+1
  swap(L[low], L[left])  fi
  return left
```

...mit dem initialen Aufruf quicksort(L, 1, length(L)).

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

0

Zum Vergleich...

... eine typische *funktionale* Realisierung von Quicksort, hier in Haskell:

```
quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (x:xs) =
  quicksort [ y | y<-xs, y<=x ] ++
  [x] ++ quicksort [ y | y<-xs, y>x ]
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

10

Vorteile funktionaler Programmierung

- *Einfach(er) zu erlernen*
...da weniger Grundkonzepte (insbesondere: keine Zuweisung, keine Schleifen, keine Sprünge)
- *Höhere Produktivität*
...da Programme dramatisch kürzer als funktional vergleichbare imperATIVE Programme (Faktor 5 bis 10)
- *Höhere Zuverlässigkeit*
...da Korrektheitsüberlegungen/-beweise einfach(er) (math. Hintergrund, keine durchscheinende Maschine)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

11

Nachteile funktionaler Programmierung

- *Geringe(re) Effizienz*
...aber: enorme Fortschritte (Effizienz oft durchaus vergleichbar mit entsprechenden C-Implementierungen), zudem Korrektheit vorrangig gegenüber Geschwindigkeit, weiters einfache(re) Parallelisierbarkeit
- *Gelegentlich unangemessen*, oft für inhärent zustandsbasierte Anwendungen oder zur GUI-Programmierung
...aber: Anwendungseignung ist stets zu überprüfen; kein Spezifikum fkt. Programmierung

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

12

Warum Haskell?

Ein Blick auf andere funktionale (Programmier-)Sprachen...

- λ -Kalkül (Ende der 30er-Jahre, Alonzo Church, Stephen Kleene)
- Lisp (frühe 60er-Jahre, John McCarthy)
- ML, SML (Mitte 70er-Jahre, Michael Gordon, Robin Milner)
- Hope (um 1980, Rod Burstall, David McQueen)
- Miranda (um 1980, David Turner)
- OPAL (Mitte der 80er-Jahre, Peter Pepper et al.)
- Haskell (Ende der 80er-Jahre, Paul Hudak, Philipp Wadler et al.)
- Göfer (Anfang der 90er-Jahre, Mark Jones)
- ...

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

13

Warum etwa nicht Haskell?

Haskell ist...

- eine fortgeschrittene moderne funktionale Sprache
 - starke Typisierung
 - lazy evaluation
 - Funktionen höherer Ordnung
 - Polymorphie/Generizität
 - pattern matching
 - Datenabstraktion (abstrakte Datentypen)
 - Modularisierung (Programmierung im Großen)
 - ...
- eine Sprache für "real world"-Probleme (<http://haskell.org/realworld/> (URL noch gültig?))
 - mächtige Bibliotheken
 - Schnittstellen z.B. zu C
 - ...

Nicht zuletzt: Wenn auch reich, ist Haskell eine "gute" Lehrsprache, auch dank Hugs!

Fassen wir noch einmal zusammen...

Steckbrief: **Funktionale Programmierung**

Grundlage: Lambda-Kalkül

Abstraktion: Funktionen (höherer Ordnung)

Eigenschaft: referentielle Transparenz

Historische Bedeutung: Basis vieler Programmiersprachen

Anwendungsbereiche: Theoretische Informatik
Artificial Intelligence
experimentelle Software
Programmierunterricht

Programmiersprachen: Lisp, ML, Miranda, Haskell, ...

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

15

sowie...

Steckbrief: **Haskell**

benannt nach: Haskell B. Curry (1900-1982)

<http://www-ggap.dcs.st-and.ac.uk/~history/Mathematicians/Curry.html>

Paradigma: rein funktionale Programmierung

Eigenschaften: lazy evaluation, pattern matching

Typsicherheit: stark typisiert, Typinferenz
modernes polymorphes Typsystem

Syntax: komprimiert, intuitiv

Informationen: <http://haskell.org>
<http://haskell.org/tutorial/>

Interpreter: Hugs (<http://haskell.org/hugs/>)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

16

Erste Schritte in Haskell

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

17

Haskell-Programme...

...gibt es in zwei (notationellen) Varianten:

Als sog.

- *(Gewöhnliches) Haskell-Skript*

Intuitiv ...alles, was nicht als Kommentar notationell ausgezeichnet ist, wird als Programmtext betrachtet.

- *Literate Haskell-Skript*

Intuitiv ...alles, was nicht als Programmtext notationell ausgezeichnet ist, wird als Kommentar betrachtet.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

18

Zur Illustration: Ein Programm als...

...(gewöhnliches) Haskell-Skript:

```
{-#####  
  FirstScript.hs ... 'ordinary scripts' erhalten  
#####-}  
-- Die konstante Funktion sum17and4  
sum17and4 :: Int  
sum17and4 = 17+4  
-- Die Funktion square zur Quadrierung einer ganzen Zahl  
square :: Int -> Int  
square n = n*n  
-- Die Funktion double zur Verdopplung einer ganzen Zahl  
double :: Int -> Int  
double n = 2*n  
-- Die Funktion doubleSquare, eine Anwendung der vorherigen  
doubleSquare :: Int  
doubleSquare = double (square (4711 - sum17and4))
```

Zum Vergleich das gleiche Programm...

...als *Iterate* Haskell-Skript:

```
#####  
FirstIterate.hs ... 'Iterate scripts' erhalten  
#####  
Die konstante Funktion sum17and4  
> sum17and4 :: Int  
> sum17and4 = 17+4  
Die Funktion square zur Quadrierung einer ganzen Zahl  
> square :: Int -> Int  
> square = n*n  
Die Funktion double zur Verdopplung einer ganzen Zahl  
> double :: Int -> Int  
> double = 2*n  
Die Funktion doubleSquare, eine Anwendung der vorherigen  
> doubleSquare :: Int  
> doubleSquare = double (square (4711 - sum17and4))
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

20

Kommentare in Haskell-Programmen

Kommentare in...

- *(gewöhnlichem) Haskell-Skript*
 - *einzeilig*: ...bis zum Rest der Zeile nach --
 - *mehrzeilig*: ...alles zwischen {- und -}

- *Iterate Haskell-Skript*

- Jede nicht durch > eingeleitete Zeile

Konvention: Datelendung...

- .hs für gewöhnliche
- .lhs für *Iterate*

Haskell-Skripte.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

21

Hugs: Der Haskell-Interpreter...

Aufruf von Hugs: `hugs <fileName>`
...und z.B. im Fall von `FirstScript` für `<fileName>` weiter mit:
`hugs> double (sum17and4)`
`42`

Wichtige Kommandos in Hugs:

```
?           Liefert Liste der Hugs-Kommandos  
:load <fileName> Lädt die Haskell-Datei <fileName> (erkennbar an Endung  
               .hs bzw. .lhs)  
:quit      Beendet den aktuellen Hugs-Lauf  
:info name Liefert Information über das mit name bezeichnete 'Objekt'  
:type exp  Liefert den Typ des Argumentausdrucks exp  
:edit <fileName>.hs Öffnet die Datei <fileName>.hs im neuesten Editor  
:find name Ermittelt die Deklaration von name im vorinstgestellten Editor  
:cows      Ausführen des Unik- oder DOS-Kommandos <cmd>
```

...mehr dazu: <http://www.haskell.org/hugs/>

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

23

Erste Schritte mit Hugs

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

22

Fehlermeldungen&Warnungen in Hugs

- Fehlermeldungen
 - Syntaxfehler
`hugs> sum17and4 == 21` ...liefert
`ERROR: syntax error in input (unexpected ')')`
 - Typfehler
`hugs> sum17and4 + False ...liefert
ERROR: Bool is not an instance of class 'Num'`
 - Programmfehler
...später
 - Modulfehler
...später
- Warnungen
 - Systemmeldungen
...später

...mehr zu Fehlermeldungen:

<http://www.cs.kent.ac.uk/people/staff/sjt/craft2e/errors.html>

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

24

Bequem...

Haskell stellt umfangreiche Bibliotheken (*Prelude.hs,...*) mit vielen vordefinierten Funktionen zur Verfügung, z.B. zum

- Umkehren von Zeichenreihen (*reverse*)
- Aufsummieren von Listenelementen (*sum*)
- Verschmelzen von Listen (*zip*)
- ...

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

25

Exkurs: Mögliche Namenskonflikte

...soll eine Funktion gleichen (bereits vordefinierten) Namens deklariert werden, können Namenskonflikte durch *Verstecken* (engl. *hiding*) vordefinierter Namen vermieden werden.

Am Beispiel von *reverse*, *sum*, *zip*:

Ergänze...

```
import Prelude hiding (reverse, sum, zip)
```

...am Anfang des Haskell-Skripts im Anschluss an die Modul-Anweisung (so vorhanden), wodurch die vordefinierten Namen *reverse*, *sum* und *zip* verborgen werden.

(Mehr dazu später im Zusammenhang mit dem Modulkonzept von Haskell).

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

26

Teil 2: Grundlagen

- Elementare Datentypen (*Bool, Int, Integer, Float, Char*)
- Tupel, Listen und Funktionen

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

27

Elementare Datentypen

...werden in der Folge nach nachstehendem Muster angegeben:

- Name des Typs
- Typische Konstanten des Typs
- Typische Operatoren (und Relatoren, so vorhanden)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

28

(Ausgew.) Elementare Datentypen (1)

Wahrheitswerte	
Typ	Bool
Konstanten	True :: Bool False :: Bool
Operatoren	&& :: Bool -> Bool -> Bool :: Bool -> Bool -> Bool not :: Bool -> Bool

	Wahrheitswerte	
	Symbol für "wahr"	
	Symbol für "falsch"	
	logisches und	
	logisches oder	
	logische Negation	

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

29

Elementare Datentypen (2)

Ganze Zahlen		
Typ	Int	Ganze Zahlen(endl. Ausschn.)
Konstanten	0 :: Int -42 :: Int 2147483647 :: Int	Symbol für "0" Symbol für "-42" Wert für "maxInt"

Operatoren	+ :: Int -> Int -> Int	Addition
	* :: Int -> Int -> Int	Multiplikation
	^ :: Int -> Int -> Int	Exponentiation
	- :: Int -> Int -> Int	Subtraktion (Infix)
	- :: Int -> Int	Vorzeichenwechsel (Prefix)
	div :: Int -> Int	Division
	mod :: Int -> Int -> Int	Divisionsrest
	abs :: Int -> Int	Absolutbetrag
	negate :: Int -> Int	Vorzeichenwechsel

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

30

Elementare Datentypen (3)

Ganze Zahlen (fortgesetzt)

Relatoren	> :: Int -> Int -> Bool	echt größer
	>= :: Int -> Int -> Bool	größer gleich
	== :: Int -> Int -> Bool	gleich
	/= :: Int -> Int -> Bool	ungleich
	<= :: Int -> Int -> Bool	keiner gleich
	< :: Int -> Int -> Bool	echt kleiner

...die Relatoren == und /= sind auf Werte aller Elementar- und vieler weiterer Typen anwendbar, beispielsweise auch auf Wahrheitswerte (Stichwort: *Überladen* (engl. *Overloading*))!

...mehr dazu später.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

31

Elementare Datentypen (4)

Ganze Zahlen (Variante)

Typ	Integer	Ganze Zahlen
Konstanten	0 :: Integer -42 :: Integer 21474836473853883234 :: Integer	Symbol für "0" Symbol für "-42" Symbol für "Große" Zahl
Operatoren	...	

...wie *Int*, jedoch ohne a priori Beschränkung für eine maximal darstellbare Zahl.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

32

Elementare Datentypen (5)

Gleitkommazahlen

Typ	Float	Gleitkommazahlen (endlicher Ausschritt)
Konstanten	0,123 :: Float -42,4711 :: Float 123,6e-2 :: Float	Symbol für "0,123" Symbol für "-42,4711" 123,6×10 ⁻²
Operatoren	+ :: Float -> Float -> Float * :: Float -> Float -> Float ... sqrt :: Float -> Float sin :: Float -> Float	Addition Multiplikation (pos.) Quadratwurzel sinus
Relatoren	== :: Float -> Float -> Bool /= :: Float -> Float -> Bool ...	gleich ungleich

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

33

Elementare Datentypen (6)

Zeichen

Typ	Char	Zeichen (Literal)
Konstanten	'a' :: Char ... 'Z' :: Char '\t' :: Char '\n' :: Char '/' :: Char '?' :: Char '\" :: Char	Symbol für "a" Symbol für "Z" Tabulator Neue Zeile Symbol für "backslash" Hochkomma Anführungszeichen
Operatoren	ord :: Char -> Int chr :: Int -> Char	Konversionsfunktion Konversionsfunktion

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

34

Zusammengesetzte Datentypen und Funktionen...

- Tupel
- Listen
 - *Spezialfall*: Zeichenreihen
- Funktionen

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

35

Tupel

Tupel ...fassen eine festgelegte Zahl von Werten möglicherweise verschiedener Typen zusammen.

~> Tupel sind *heterogen!*

Beispiele:

- ...Modellierung von Studentendaten
(`"Max Mustermann", "e0123456@studat.tu.wien.ac.at", 534`) :: (String, String, Int)
- ...Modellierung von Bibliotheksdaten
(`"PeytonJones", "Implementing Funct. Lang", 1987, True`) :: (String, String, Int, Bool)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

36

Tupel...

- Allgemeines Muster

```
(v1,v2,...,vk) :: (T1,T2,...,Tk)
```

mit `v1,...,vk` Bezeichnungen von Werten und `T1,...,Tk` Bezeichnungen von Typen mit

```
v1 :: T1, v2 :: T2, ..., vk :: Tk
```

Lies: `v1` ist vom Typ `T1`

- Standardkonstruktor
(. , . . . , . . . , .)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

37

Spezialfall: Paare ("Zweitupel")

- Beispiele

```
type Point = (Float, Float)
```

```
(0,0,0.0) :: Point  
(3,14,17.4) :: Point
```

- Standardselektoren (für Paare)

```
fst (x,y) = x  
snd (x,y) = y
```
- Anwendung der Standardselektoren

```
fst (0,0,0.0) = 0.0  
snd (3,14,17.4) = 17.4
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

38

Hilfreich...

Typsynonyme

```
type Student = (String, String, Int)  
type Buch = (String, String, Int, Bool)
```

...erhöhen die Transparenz in Programmen.

Wichtig: Typsynonyme definieren *keine* neuen Typen, sondern einen Namen für einen schon existierenden Typ (später mehr dazu).

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

39

Tupel...

Selbstdefinierte Selektorfunktionen...

```
type Student = (String, String, Int)
```

```
name :: Student -> String  
email :: Student -> String  
kennzahl :: Student -> Int
```

```
name (n,e,k) = n  
email (n,e,k) = e  
kennzahl (n,e,k) = k
```

...mittels *Mustererkennung* (engl. *pattern matching*) (später mehr dazu).

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

40

Selbstdefinierte Selektorfunktionen...

Ein weiteres Beispiel...

```
autor :: Buch -> String
kurzTitel :: Buch -> String
erscheinungsjahr :: Buch -> Int
ausgeliehen :: Buch -> Bool
```

```
autor (a,t,j,b) = a
kurzTitel (a,t,j,b) = t
erscheinungsjahr (a,t,j,b) = j
ausgeliehen (a,t,j,b) = b
```

```
autorLehnt (a,t,j,b) = (autor (a,t,j,b), ausgeliehen (a,t,j,b))
```

... auch hier mittels Mustererkennung

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 41

Listen

Listen ...fassen eine beliebige/unbestimmte Zahl von Werten gleichen Typs zusammen.

↪ Listen sind *homogen!*

Einfache Beispiele:

- Listen ganzer Zahlen
[2,5,12,42] :: [Int]
- Listen von Wahrheitswerten
[True,False,True] :: [Bool]
- Listen von Gleitkommazahlen
[3,14,5.0,0.12,21] :: [Float]
- Leere Liste
[]
- ...

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 42

Listen

Beispiele komplexerer Listen:

- Listen von Listen
[[2,4,23,2,5],[3,4],[56,7,6,1]] :: [[Int]]
- Listen von Paaren
[(3.14,42.0),(56.1,51.3)] :: [(Float,Float)]
- ...
- *Ausbl./K:* Listen von Funktionen
[fac, abs, negate] :: [Integer -> Integer]

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 43

Vordefinierte Funktionen auf Listen

Die Funktion `length` mit einigen Aufrufen:

```
length :: [a] -> Integer
length [] = 0
length (x:xs) = 1 + length xs

length [1, 2, 3] => 3
length ['a','b','c'] => 3
length [[1],[2],[3]] => 3
```

Die Funktionen `head` und `tail` mit einigen Aufrufen:

```
head :: [a] -> a
head (x:xs) = x

tail :: [a] -> [a]
tail (x:xs) = xs

head [[1],[2],[3]] => [1]
tail [[1],[2],[3]] => [[2],[3]]
```

Spezielle Notationen für Listen

- Spezialfälle (i.w. für Listen über Zahlen und Zeichen)
...[2..6] kurz für [2,3,4,5,6]
...[11,9..2] kurz für [11,9,7,5,3]
...['a','d'..'j'] kurz für ['a','d','e','f','g','h','i','j']
...[0.0,0.3..1.0] kurz für [0.0,0.3,0.6,0.9]
- *Listenkomprehension*
...ein erstes Beispiel:
[3*n | n <- list] kurz für [3,6,9,12], wobei hier list vom Wert [1,2,3,4] vorausgesetzt ist.
↪ Listenkomprehension ist ein sehr elegantes und ausdruckskräftiges Sprachkonstrukt!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 45

Zeichenreihen

...in Haskell als spezielle Listen realisiert:

Typ	String	Zeichenreihen Deklaration (als Liste von Zeichen)
Typ	String	Zeichenr. für "Haskell"
Konstanten	"Haskell" :: String	Leere Zeichenreihe ""
Operatoren	++ :: String -> String -> String	Konkatenation
Relatoren	== :: String -> String -> Bool /= :: String -> String -> Bool	gleich ungleich

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 46

Weitere Beispiele zu Zeichenreihen

```
['h','e','l','l','o'] == "hello"
"hello" ++ " world" == "hello world"
```

Es gilt:

```
[1,2,3] == 1:2:3:[]
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 47

Funktionen in Haskell

...am Beispiel der Fakultätsfunktion:

Zur Erinnerung:

$$i : \mathbb{N} \rightarrow \mathbb{N}$$
$$n! = \begin{cases} 1 & \text{falls } n = 0 \\ n * (n - 1)! & \text{sonst} \end{cases}$$

...und eine mögliche Realisierung in Haskell:

```
fac :: Integer -> Integer
fac n = if n == 0 then 1 else (n * fac(n-1))
```

Beachte: ...Haskell stellt eine Reihe, oft eleganterer, notativer Varianten zur Verfügung!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 48

Fkt. in Haskell: Notat. Varianten (1)

...am Beispiel der Fakultätsfunktion.

```
fac :: Integer -> Integer
```

(1) In Form "bedingter Gleichungen"

```
fac n
| n == 0    == 1
| otherwise == n * fac (n - 1)
```

~> *Hinweis:* Variante (1) ist "der" Regelfall in Haskell!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

49

Fkt. in Haskell: Notat. Varianten (1)

(2) λ -artig

```
fac = \n -> (if n == 0 then 1 else (n * fac (n - 1)))
```

- Reminiszenz an den funktionalen Programmierung zugrundeliegenden λ -Kalkül ($\lambda x.y. (x + y)$)

- In Haskell: $\lambda x.y \rightarrow x+y$ sog. *anonyme* Funktion. Praktisch, wenn der Name keine Rolle spielt und man sich deshalb bei Verwendung *anonymer* Funktionen keinen zu überlegen braucht.

(3) *Gleichungsorientiert*

```
fac n = if n == 0 then 1 else (n * fac (n - 1))
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

50

Fkt. in Haskell: Notat. Varianten (2)

...am Beispiel weiterer Funktionen.

```
kVA :: Float -> (Float, Float)
```

-- Berechnung von Volumen (V) und Fläche (A) einer Kugel (K).

Zur Erinnerung: $V = \frac{4}{3} \pi r^3$ $A = 4 \pi r^2$

Mittels *lokaler Deklarationen*...

(4a) *where*-Konstrukt

```
kVA r =
  ((4/3) * myPi * rcube r, 4 * myPi * square r)
  where
    myPi    = 3.14
    rcube x = x * square x
    square x = x * x * x
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

51

Fkt. in Haskell: Notat. Varianten (3)

bzw...

(4b) *let*-Konstrukt

```
kVA r =
  let
    myPi    = 3.14
    rcube x = x * square x
    square x = x * x * x
  in
    ((4/3) * myPi * rcube r, 4 * myPi * square r)
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

52

Fkt. in Haskell: Notat. Varianten (4)

In einer Zeile...

(5a) ...mittels ""

```
kVA r =
  ((4/3) * myPi * rcube r, 4 * myPi * square r)
  where
    myPi = 3.14; rcube x = x * square x; square x = x * x
    (5b) ...mittels ""
    kVA r =
      let myPi = 3.14; rcube x = x * square x; square x = x * x
      in
        ((4/3) * myPi * rcube r, 4 * myPi * square r)
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

53

Fkt. in Haskell: Notat. Varianten (5)

Spezialfall: *binäre* (zweistellige) Funktionen...

```
imax :: Integer -> Integer -> Integer
imax p q
| p >= q    = p
| otherwise = q

tripleMax :: Integer -> Integer -> Integer
tripleMax p q r
| (imax p q == p) && (p 'imax' r == p)    = p
| ...                                     = r
| otherwise                               = r

...imax in tripleMax als Präfix- und als Infixoperator verwandt
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

54

Fkt. in Haskell: Notat. Varianten (6)

Musterbasieret...

```
fib :: Integer -> Integer
```

```
fib 0 = 1
```

```
fib 1 = 1
```

```
fib n = fib(n-2) + fib(n-1)
```

```
capVowels :: Char -> Char
```

```
capVowels 'a' = 'A'
```

```
capVowels 'e' = 'E'
```

```
capVowels 'i' = 'I'
```

```
capVowels 'o' = 'O'
```

```
capVowels 'u' = 'U'
```

```
capVowels c = c
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

55

Fkt. in Haskell: Notat. Varianten (7)

Mittels case-Ausdrucks...

```
capVowels :: Char -> Char
capVowels letter
  = case letter of
    'a' -> 'A'
    'e' -> 'E'
    'i' -> 'I'
    'o' -> 'O'
    'u' -> 'U'
    letter -> letter

decapVowels :: Char -> Char
decapVowels letter
  = case letter of
    'A' -> 'a'
    'E' -> 'e'
    'I' -> 'i'
    'O' -> 'o'
    'U' -> 'u'
    otherwise -> letter
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

56

Fkt. in Haskell: Notat. Varianten (8)

Mittels *Muster* und "wild cards"...

```
add :: Integer -> Integer -> Integer
add n 0 = n
add 0 n = n
add m n = m+n

mult :: Integer -> Integer -> Integer
mult _ 0 = 0
mult 0 _ = 0
mult m n = m*n
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

57

Muster können (u.a.) sein...

- *Werte* (z.B. 0, 'c', True) ...ein Argument "passt" auf das Muster, wenn es vom entsprechenden Wert ist.
 - *Variablen* (z.B. n) ...jedes Argument passt.
 - *Wild card* "_" ...jedes Argument passt (sinvoll für nicht zum Ergebnis beitragende Argumente)
 - ...
- ~> mehr über Muster und musterbasierte Funktionsdefinitionen später...

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

58

Literaturhinweis

...auf den Haskell-Sprachreport:

- *Haskell 98: Language and Libraries. The Revised Report.* Simon Peyton Jones (Hrsg.). Cambridge University Press, 2003.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

59

...die uns zu folgenden Thema führen

Mehr zu allg. Grundlagen und zu Haskell, insbesondere zu...

- Funktionen
 - ...und darüber wie man sie definieren/notieren kann
 - ~> Notationelle Alternativen (bereits besprochen)
 - ~> **Funktionssignaturen, Funktionsausdrücke, Klammersparungsregeln**
 - ~> Abschtsregel und Layout-Konventionen
 - und in der Folge weiterführend zu
 - Klassifikation von Rekursionstypen
 - Anmerkungen zu Effektivität und Effizienz
 - Komplexitätsklassen

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

61

Funktionen und ihre Signaturen (1)

Zur *Veranschaulichung, noch konkreter*:

Die Signaturen der Funktionen **f**

```
f :: Int -> (Int -> Int)
```

(aufgrund der Klammersparungsregeln gleichbedeutend mit der ungeklammerten Kurzform **f :: Int -> Int -> Int**) und **g**

```
g :: (Int -> Int) -> Int
```

sind *grundsätzlich verschieden* und *unbedingt auseinanderzuhalten!*

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

63

Fortf. von Kap. 1, Teil 2, Grundlagen

Motivation...

Betrachte folgendes Exzerpt von Aufgabenblatt 1:

"Ein einfacher Editor kann in Haskell wie folgt realisiert werden:

```
type Editor = [Char]
```

Schreiben Sie eine Haskell-Rechenvorschrift *ersetze* mit der Signatur

```
ersetze :: Editor -> Int -> String -> String -> Editor
```

die angesetzt auf..."

Zwei naheliegende Fragen...

- Warum so wenige Klammern?
- Warum so viele Pfeile (->) und warum so wenige Kreuze (×)? Warum nicht folgende Signaturzeile?
"ersetze :: (Editor x Int x String x String) -> Editor"
Beachte: Haskell-Korrekt (d.h. "_" statt x) wäre
ersetze :: (Editor, Int, String, String) -> Editor

Klammersparungsregeln in Funktionssignaturen

Konvention (von essentieller Bedeutung):

- Der *Typkonstruktor* → ist *rechtsassoziativ!*

Das bedeutet:

- Die Funktionssignatur

```
f :: Int -> Float -> Int -> String -> Char
```

steht abkürzend für

```
f :: (Int -> (Float -> (Int -> (String -> Char))))
```

- Wann immer eine abweichende Klammerung intendiert ist, *musst* explizit geklammert werden!
(vgl. *Klammersparungsregeln bei arithmetischen Ausdrücken*)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

62

Funktionen und ihre Signaturen (2)

Warum?

- **f** ist eine Funktion, die ganze Zahlen auf Abbildungen ganzer Zahlen in sich abbildet.
- **g** ist eine Funktion, die Abbildungen ganzer Zahlen in sich auf ganze Zahlen abbildet.

Zur Übung:

- Überlegen Sie sich, ob die Funktion **+** (Addition auf ganzen Zahlen) dem Signaturschema von **f** oder dem von **g** folgt.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

64

Funktionen und ihre Signaturen (3)

Ein weiteres Beispiel, noch konkreter und noch ein wenig komplexer...

Mit folgenden Deklarationen für f und g

```
f :: Int -> (Int -> Int)
f 1 = (+)
f 2 = (-)
f 3 = (*)
f _ = div
```

```
g :: (Int -> Int -> Int) -> Int
g h = h 6 3
```

...liefern die nachstehenden Aufrufe von f und g die angegebenen Resultate:

```
Main> f 1 2 3           kurz fuer: (((f 1) 2) 3)
```

```
5
```

```
Main> f 3 2 3
```

```
6
```

```
Main> g (+)
```

```
9
```

```
Main> g (*)
18
```

Funktionen und ihre Signaturen (4)

Offenbar gilt...: in g sind die Argumente 6 und 3 fest vorgegeben. Betrachte deshalb jetzt die folgende "Erweiterung" k von g , die das vermeidet:

```
k :: (Int -> Int -> Int) -> Int -> Int
k h x y = h x y
```

Beachte: ...aufgrund der Klammersparungsregeln für Funktionsterme (Inkassoziativ) und \rightarrow (rechtsassoziativ) steht obige Deklaration von k abkürzend für:

```
k :: ((Int -> (Int -> Int)) -> (Int -> (Int -> Int)))
((k h) x) y = ((h x) y)
```

Für k sind jetzt folgende Aufrufe mit variablen Argumenten möglich:

```
Main> k (*) 3 5
15
Main> k (+) 17 4
21
Main> k div 42 5
8
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 66

Funktionen und ihre Signaturen (5)

Zur Übung:

Vergleichen Sie die Deklaration der Funktion f

```
f :: Int -> (Int -> Int -> Int)
f 1 = (+)
f 2 = (-)
f 3 = (*)
f _ = div
```

...mit der Deklaration ihrer scheinbar naheliegenden "dualen Variante" g :

```
g :: (Int -> Int -> Int) -> Int
g (+) = 1
g (-) = 2
g (*) = 3
g div = 42
g _ = 99
```

- Was beobachten Sie, wenn Sie die Funktionen f und g aufrufen?
- Haben Sie (schon) eine Erklärung dafür?

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 67

Funktionen und ihre Signaturen (6)

Bleiben Sie auch an folgender Frage dran...

- Warum möglicherweise sind die Klammersparungsregeln für \rightarrow zugunsten der Rechtsassoziativität von \rightarrow

```
f :: (Int -> (Int -> Int))
und nicht der Linksassoziativität gefallen?
f :: ((Int -> Int) -> Int) -> Int
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 68

Funktionen und ihre Signaturen (7)

In jedem Falle gilt:

Die Einsicht in den Unterschied

- von

```
f :: Int -> Int -> Int -> Int
```

...aufgrund der Rechtsassoziativität von \rightarrow abkürzend und gleichbedeutend mit der vollständig, aber nicht überflüssig geklammerten Version

```
f :: (Int -> (Int -> (Int -> Int)))
```

- und von

```
f :: (((Int -> Int) -> Int) -> Int)
```

ist *essentiell* und von *absolut zentraler* Bedeutung!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 69

Funktionen und ihre Signaturen (8)

Bewusst pointiert...

Ohne diese Einsicht ist erfolgreiche Programmierung (speziell) im funktionalen Paradigma

- nicht möglich
- oder allenfalls Zufall!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 70

Bestandsaufnahme (1)

- Bis jetzt:
 - ...Konzentration auf Funktionsdeklarationen und ihre Signaturen bzw. Typen
- Ab jetzt:
 - ...Konzentration auf Funktionsterme und ihre Signaturen bzw. Typen

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 71

Bestandsaufnahme (2)

Tatsache:

Wir sind gewohnt, mit Ausdrücken der Art

```
add 2 3
```

umzugehen. (Auch wenn wir gewöhnlich 2+3 statt `add 2 3` schreiben.)

Frage:

- Warum könnte es sinnvoll sein, auch mit (scheinbar unvollständigen) Ausdrücken wie `add 2` umzugehen?
- Entscheidend für die Antwort: Können wir einen Ausdruck wie `add 2` sinnvoll eine Bedeutung geben und wenn ja, welche?

Funktionsterme und ihre Typen (1)

Betrachten wir die Funktion `add` zur Addition ganzer Zahlen noch einmal im Detail:

```
add :: Int -> Int -> Int
add m n = m+n
```

abkuerzend fuer: $((add\ m)\ n) = m+n$

Dann sind die Ausdrücke `add`, `add 2` und `add 2 3` von den Typen:

```
add :: Int -> Int -> Int
add 2 :: Int -> Int
add 2 3 :: Int
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

73

Funktionsterme und ihre Typen (2)

Erinnerung:

```
add :: Int -> Int -> Int
entspricht wg. vereinbarter Rechtsassoziativität von ->
add :: Int -> (Int -> Int)
```

Somit *verbal* umschrieben:

- `add :: Int -> Int -> Int`
...bezeichnet eine Funktion, die ganze Zahlen auf Funktionen von ganzen Zahlen in ganze Zahlen abbildet (Rechtsassoziativität von \rightarrow !).
- `add 2 :: Int -> Int`
...bezeichnet eine Funktion, die ganze Zahlen auf ganze Zahlen abbildet.
- `add 2 3 :: Int`
...bezeichnet eine ganze Zahl (nämlich 5).

Funktionsterme und ihre Typen (3)

Damit haben wir eine Antwort auf unsere Ausgangsfrage...

- Warum könnte es sinnvoll sein, auch mit (*schreibbar unvollständigen*) Ausdrücken wie
`add 2`
umzugehen?
- Entscheidend für die Antwort: Können wir einem Ausdruck wie `add 2` sinnvoll eine Bedeutung geben und wenn ja, welche?

Nämlich:

Es ist sinnvoll, mit Ausdrücken der Art `add 2` umzugehen, weil

- wir ihnen sinnvoll eine Bedeutung zuordnen können!
- im Falle von `add 2`:
...`add 2` bezeichnet eine Funktion auf ganzen Zahlen, die angewendet auf ein Argument dieses Argument um 2 erhöht als Resultat liefert.

Funktionsterme und ihre Typen (5)

Beachte:

Der Ausdruck (Funktionsterm)

```
k add 2 3
```

steht kurz für

```
((k add) 2) 3)
```

Analog stehen die Ausdrücke (Funktionsterme)

```
k add
k add 2
```

kurz für

```
(k add)
((k add) 2)
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

77

Funktionsterme und ihre Typen (4)

Betrachte auch folgendes Beispiel von vorhin unter dem neuen Blickwinkel auf Funktionsterme und ihre Typen:

```
k :: (Int -> Int -> Int) -> Int -> Int -> Int
k h x y = h x y
```

Dann gilt:

```
k :: (Int -> Int -> Int) -> Int -> Int -> Int
k add :: Int -> Int -> Int
k add 2 :: Int -> Int
k add 2 3 :: Int
```

Zur Übung:

- Ausprobieren! In Hugs lässt sich mittels des Kommandos `:t <Ausdruck>` der Typ eines Ausdrucks bestimmen!
Bsp.: `:t k add 2` liefert `k add 2 :: Int -> Int`

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

76

Funktionsterme und ihre Typen (6)

Beobachtung (anhand des vorigen Beispiels):

- Funktionen in Haskell sind grundsätzlich *einstellig!*
- Wie die Funktion `k` zeigt, kann dieses Argument komplex sein, bei `k` z.B. eine Funktion, die ganze Zahlen auf Funktionen ganzer Zahlen in sich abbildet.

Beachte:

Die Sprechweise, Argument der Funktion `k` sei eine zweistellige Funktion auf ganzen Zahlen, ist *lax* und *unpräzise*, gleichwohl (aus Gründen der Einfachheit und Bequemlichkeit) üblich.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

78

Funktionsterme und ihre Typen (8)

Beispiele:

- *Keine Klammerung* (\rightsquigarrow Konvention greift!)

```
f :: Int -> Tree -> Graph -> ...
```

`f` ist einstellige Funktion auf ganzen Zahlen, nämlich `Int`, die diese abbildet auf...

- *Explizite Klammerung* (\rightsquigarrow Konvention aufgehoben, wo gewünscht!)

```
f :: (Int -> Tree) -> Graph -> ...
```

`f` ist einstellige Funktion auf Abbildungen von ganzen Zahlen auf Bäume, nämlich `Int -> Tree`, die diese abbildet auf...

Hinweis: Wie wir Bäume und Graphen in Haskell definieren können, lernen wir bald.

Funktionsterme und ihre Typen (9)

Auch noch zu...

- ...
- Wann immer dies nicht erwünscht ist, muss dies durch explizite Klammern in der Funktionsnatur erzwingen werden.

Beispiele:

- *Keine Klammerung*

```
f :: Int -> Tree -> Graph -> ...
```

f ist einstellige Funktion auf ganzen Zahlen, nämlich Int, die diese abbildet auf...

- *Explizite Klammerung*

```
f :: (Int,Tree) -> Graph -> ...
```

f ist einstellige Funktion auf Paaren aus ganzen Zahlen und Bäumen, nämlich (Int,Tree), die diese abbildet auf...

Funktionsterme und ihre Typen (10)

Noch einmal zurück zum Beispiel der Funktion k:

```
k :: (Int -> Int -> Int) -> Int -> Int -> Int
```

...k ist eine einstellige Funktion, die eine zweistellige Funktion auf ganzen Zahlen als Argument erwartet (kx1) und auf eine Funktion abbildet, die ganze Zahlen auf Funktionen ganzer Zahlen in sich abbildet.

Zur Deutlichkeit die Signatur von k auch noch einmal vollständig, aber nicht überflüssig geklammert:

```
k :: ((Int -> (Int -> Int)) -> (Int -> (Int -> Int)))
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

82

Funktionsterme und ihre Typen (11)

Das Beispiel von k fortgesetzt:

```
k add :: Int -> Int -> Int
```

...k add ist eine einstellige Funktion, die ganze Zahlen auf Funktionen ganzer Zahlen in sich abbildet.

Zur Deutlichkeit auch hier noch einmal vollständig, aber nicht überflüssig geklammert:

```
(k add) :: (Int -> (Int -> Int))
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

83

Funktionsterme und ihre Typen (12)

Das Beispiel von k weiter fortgesetzt:

```
k add 2 :: Int -> Int
```

...k add 2 ist eine einstellige Funktion, die ganze Zahlen in sich abbildet.

Zur Deutlichkeit auch hier wieder vollständig, aber nicht überflüssig geklammert:

```
((k add) 2) :: (Int -> Int)
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

84

Funktionsterme und ihre Typen (13)

Das Beispiel von k abschließend fortgesetzt:

```
k add 2 3 :: Int
```

k add 2 3 bezeichnet ganze Zahl: in diesem Falle 5.

Zur Deutlichkeit auch dieser Funktionsterm vollständig, aber nicht überflüssig geklammert:

```
((k add) 2) 3) :: Int
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

85

Wichtige Vereinbarungen in Haskell

Wenn in Haskell durch Klammerung nichts anderes ausgedrückt wird, gilt für

- Funktionssignaturen *Rechtsassoziativität*, d.h.

```
k :: (Int -> Int -> Int) -> Int -> Int -> Int
```

steht für

```
k :: ((Int -> (Int -> Int)) -> (Int -> (Int -> Int)))
```

- Funktionsterme *Linksassoziativität*, d.h.

```
k add 2 3 :: Int
```

steht für

```
((k add) 2) 3) :: Int
```

als vereinbart!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

86

Zum Abschluss des Signaturthemas (1)

Frage:

- Warum mag uns ein Ausdruck wie

```
add 2
```

"unvollständig" erscheinen?

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

87

Zum Abschluss des Signaturthemas (2)

...Weil wir im Zusammenhang mit der Addition tatsächlich weniger an Ausdrücke der Form

```
add 2 3
```

als vielmehr an Ausdrücke der Form

```
add' (2,3)
```

gewohnt sind!

Erinnern Sie sich?

```
+ : Z × Z → Z
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

88

Zum Abschluss des Signaturthemas (3)

Der Unterschied liegt in den Signaturen der Funktionen `add` und `add'`:

```
add :: Int -> (Int -> Int)
add' :: (Int,Int) -> Int
```

Mit diesen Signaturen von `add` und `add'` sind einige Beispiele...

- *korrekter* Aufrufe:

```
add 2 3      => 5 :: Int
add' (2,3)   => 5 :: Int
add 2        :: Int -> Int
```

- *Inkorrekter* Aufrufe:

```
add (2,3)
add' 2 3      -- beachte: add' 2 3 steht kurz fuer (add' 2) 3
add' 2
```

Zum Abschluss des Signaturthemas (4)

Mithin...

- ...die Funktionen `+` und `add'` sind echte *zweistellige* Funktionen
wohingegen...
- ...die Funktion `add` einstellig ist und nur aufgrund der Klammerungsregeln scheinbar ebenfalls "zweistellige" Aufrufe zulässt:
`add 17 4`

Aber: `add 17 4` steht kurz für `(add 17) 4`. Die geklammerte Variante macht deutlich: Ein Argument nach dem anderen und nur eines zur Zeit...

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 90

Fazit zum Signaturthema (1)

Wir müssen nicht nur sorgfältig

- *zwischen*

```
f :: Int -> Int -> Int
```

...aufgrund der *Rechtsassoziativität* von \rightarrow abkürzend und gleichbedeutend ist mit

```
f :: Int -> (Int -> Int)
```

- und

```
f :: (Int -> Int) -> Int
```

unterscheiden, sondern ebenso sorgfältig auch

- *zwischen*

```
f :: (Int,Int) -> Int
```

- und

```
f :: Int -> (Int,Int)
```

und nicht zuletzt zwischen allen vier Varianten insgesamt!

Fazit zum Signaturthema (2)

Mithin, schreiben Sie

```
f :: Int -> Int -> Int
```

nur, wenn Sie auch wirklich

```
f :: Int -> (Int -> Int)
```

meinen und nicht etwa

```
f :: (Int -> Int) -> Int
```

oder

```
f :: (Int,Int) -> Int
```

oder

```
f :: Int -> (Int,Int)
```

Es macht einen Unterschied!

Und deshalb die Bitte:

- Gehen Sie die vorausgegangenen Beispiele noch einmal Punkt für Punkt durch und vergewissern Sie sich, dass Sie sie im Detail verstanden haben.

Das ist wichtig, weil...

- dieses Verständnis und der aus diesem Verständnis heraus mögliche kompetente und selbstverständliche Umgang mit komplexen Funktionssignaturen und Funktionstermen essentiell für alles weitere ist!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 93

Ein kurzer Ausblick

Wir werden auf die Unterschiede und die Vor- und Nachteile von Deklarationen in der Art von

```
add :: Int -> (Int -> Int)
```

und

```
add' :: (Int,Int) -> Int
```

im Verlauf der Vorlesung unter den Schlagwörtern *Funktionen höherer Ordnung*, *Currfizierung*, *Funktionen als "first class citizens"* wieder zurückkommen.

Behalten Sie die Begriffe im Hinterkopf und blättern Sie zu gegebener Zeit in Ihren Unterlagen wieder hierher zurück.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 94

Ergänzungen zu Funktionstermen (1)

Betrachten wir noch einmal die Funktion `add`:

```
add :: Int -> (Int -> Int)
```

```
add m n = m+n
```

...und die Frage nach der "Existenz(berechtigung)" von

```
add 2 :: Int -> Int
```

...welches eine Funktion auf ganzen Zahlen ist, die ihr um 2 erhöhtes Argument als Resultat liefert.

Wir können diese Funktion `doubleInc` nennen...

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 95

Ergänzungen zu Funktionstermen (2)

...und in natürlicherweise wie folgt definieren:

```
doubleInc :: Int -> Int
doubleInc n = 2+n
```

Wir können die Definition von `doubleInc` aber auch auf die Funktion `(add 2)` abstützen:

```
doubleInc :: Int -> Int
doubleInc n = (add 2) n
```

...oder noch kürzer argumentlos (als Identität von Funktionen) einführen:

```
doubleInc :: Int -> Int
doubleInc = (add 2)
```

Beobachtung: `doubleInc` ist (nur noch) ein anderer Name für die Funktion `(add 2)`, die hier und in den obigen Bsp. nur der Deutlichkeit halber geklammert ist.

Ergänzungen zu Funktionstermen (3)

Vergleiche `doubleInc`, `add 2`

```
doubleInc :: Int -> Int
doubleInc = add 2
```

mit

```
\n -> add 2 n
```

Beobachtung: `doubleInc`, `add 2` und `\n -> add 2 n` sind...

- i.w. gleichwertige Formulierungen derselben Funktion
- i.w. dadurch unterschieden, dass `doubleInc` eine herkömmlich und im gewohnten Sinn benannte Funktion ist, wohingegen `add 2` und `(\n -> (add 2) n)` unbenannt, zumindest nicht im gewohnten Sinn mit einem Namen benannt sind: die Funktion `(\n -> (add 2) n)` speziell ist im Haskell-Jargon eine sog. *anonyme Funktion!*

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

97

“Erfahrenheits”-Faustregel

Die Implementierung einer Funktion wie `doubleInc`

- durch

```
doubleInc :: Int -> Int
doubleInc n = 2*n
```

...deutet darauf hin, dass vermutlich noch wenig Erfahrung mit funktionaler Programmierung vorliegt

- durch

```
doubleInc :: Int -> Int
doubleInc = (+) 2
```

...deutet darauf hin, dass vermutlich bereits mehr Erfahrung mit funktionaler Programmierung vorliegt

- durch

```
\n -> 2*n
```

...deutet gleichfalls darauf hin, dass schon mehr Erfahrung mit funktionaler Programmierung vorliegt, und darüberhinaus, dass in der konkreten Anwendungssituation ein Name, unter dem auf die Funktion mit der Bedeutung “doubleInc” zugegriffen werden könnte, keine Rolle spielt.

Als Ausblick... (1)

...ein kleines Beispiel schon jetzt:

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

Anwendung:

```
map (\n -> 2+n) [1,2,3] => [3,4,5]
```

...oder genauso gut

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

Machen Sie sich klar, dass die Typisierung von `add` folgendes nicht zulässt:
`map (add 2) [1,2,3]`

~> später mehr dazu unter dem Stichwort “Funktionale”, speziell Funktionale auf Listen...

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

100

Ein weiterer Nachtrag: Operatoren in Haskell

Operatoren in Haskell sind...

- ...grundsätzlich *Präfixoperatoren*, insbesondere alle selbst-deklarierten Operatoren (*wilgo*: selbstdeklarierte Funktionen)
- Beispiele:* `fac 5`, `imax 2 3`, `triplexMax 2 5 3`,...
- ...in einigen wenigen Fällen *Infixoperatoren*, dies gilt insbesondere für arithmetische Operatoren
Beispiele: `2+3`, `3*5`, `7-4`, `5^3`,...

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

101

Binäre Operatoren in Haskell: Infix- vs. Präfix

Für binäre Operatoren in Haskell gilt...

- Binäre Operatoren `bop`, die standardmäßig als...
 - Präfixoperatoren verwendet werden, können in der Form `'bop'` als Infixoperator verwendet werden
Beispiel: `2 'imax' 3` (statt standardmäßig `imax 2 3`)
 - Infixoperatoren verwendet werden, können in der Form `(bop)` als Präfixoperator verwendet werden
Beispiel: `(+) 2 3` (statt standardmäßig `2+3`)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

102

Abschließend zu Funktionstermen (1)

Betrachten wir noch einmal die Funktionen `add` und `add'`:

```
add :: Int -> (Int -> Int)
```

```
add' :: (Int,Int) -> Int
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

103

Abschließend zu Funktionstermen (2)

...hier noch einmal zusammen mit ihren Implementierungen:

```
add :: Int -> (Int -> Int)
add m n = m+n
```

```
add' :: (Int,Int) -> Int
add' (m,n) = m+n
```

Sprechweise: Die Funktion...

- `add` ist *curryfiziert*
- `add'` ist *uncurryfiziert*

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

104

Curryfiziert vs. uncurryfiziert (1)

Idee: ...ziehe die Art der Konsumation mehrerer Argumente zur Klassifizierung von Funktionen heran

Erfolgt die Konsumation mehrerer Argumente durch Funktionen...

- einzeln Argument für Argument: *curryfiziert*
- gebündelt als Tupel: *uncurryfiziert*

Beispiele:

Funktion *add* curryfiziert: *add* 2 3 bzw. (*add* 2) 3
Funktion *add'* uncurryfiziert: *add'* (2,3)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

105

Curryfiziert vs. uncurryfiziert (2)

Zentral sind die beiden *Funktionale* (Synonym: *Funktionen höherer Ordnung*) *curry* und *uncurry*...

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

```
uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry g (x,y) = g x y
```

Intuitiv:

- *Curryfizieren* ersetzt Produkt-/Tupelbildung "x" durch Funktionspfeil "→".
- *Decurryfizieren* ersetzt Funktionspfeil "→" durch Produkt-/Tupelbildung "x".

Bemerkung: Die Bezeichnung geht auf Haskell B. Curry zurück, die (weit ältere) Idee auf M. Schönfinkel aus der Mitte der 20er-Jahre.

Curryfiziert vs. uncurryfiziert (3)

Die Funktionale *curry* und *uncurry* bilden...

- uncurryfiziert vorliegende Funktionen auf ihr curryfiziertes Gegenstück *ab*, d.h.
...für uncurryfiziertes *f* :: (a,b) -> c ist
curry f :: a -> b -> c curryfiziert.
- curryfiziert vorliegende Funktionen auf ihr uncurryfiziertes Gegenstück *ab*, d.h.
...für curryfiziertes *g* :: (a -> b -> c) ist
uncurry g :: (a,b) -> c uncurryfiziert.

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

uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry g (x,y) = g x y
uncurry g :: (a,b) -> c
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

108

Curryfiziert oder uncurryfiziert?

...das ist hier die Frage.

Zum einen...

- Geschmackssache (sozusagen eine notationale Spielerei)
...sicher, auch das, aber: die Verwendung curryfizierter Formen ist in der Praxis vorherrschend
~> *f x*, *f x y*, *f x y z*,... möglicherweise eleganter als
f x, *f (x,y)*, *f (x,y,z)*,...?

Zum anderen (und weit wichtiger!) folgendes...

- Sachargument
... (nur) Funktionen in curryfizierter Darstellung unterstützen *partielle Auswertung*
~> Funktionen liefern Funktionen als Ergebnis!

Beispiel: *add* 4711 :: Int -> Int

...ist eine einstellige Funktion auf den ganzen Zahlen, die ihr Argument um 4711 erhöht als Resultat zurückliefert.

Absetsregel (engl. offside rule) (1)

...layout-abhängige Syntax als notationelle Besonderheit in Haskell

"Absets"-Regel...

- Erstes Zeichen einer Deklaration (bzw. nach *let*, *where*):
...*Startspalte neuer "Box" wird festgelegt*
- Neue Zeile...
 - gegenüber der aktuellen Box nach rechts eingerückt:
...*aktuelle Zeile wird fortgesetzt*
 - genau am linken Rand der aktuellen Box: ...*neue Deklaration wird eingeleitet*
 - weiter links als die aktuelle Box: ...*aktuelle Box wird beendet ("Absetsituation")*

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

111

Ein Beispiel zur Absetsregel (1)

Unsere Funktion *kVA* zur Berechnung von Volumen und Oberfläche einer Kugel mit Radius *r*:

```
kVA r =
  ((4/3) * myPi * rcube r, 4 * myPi * square r)
  where
    myPi = 3.14
    rcube x = x * x *
    square x
```

```
square x = x * x
```

...nicht schön, aber korrekt. Das Layout genügt der Absetsregel von Haskell und damit den Layout-Konventionen.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

112

Layout-Konventionen für Haskell-Programme

Für die meisten gängigen Programmiersprachen gilt:

- Das Layout eines Programms hat Einfluss
 - auf seine Leslichkeit, Verständlichkeit, Wartbarkeit
 - aber nicht auf seine Bedeutung

Für Haskell gilt das nicht!

- Das Layout eines Programms trägt in Haskell Bedeutung!
- Reminiszenz an Cobol, Fortran. Layoutabhängigkeit aber auch zu finden in modernen Sprachen wie z.B. *occam*.
- Für Haskell ist für diesen Aspekt des Sprachentwurfs eine grundsätzlich andere Entwurfsentscheidung getroffen worden als z.B. für Java, Pascal, C, etc.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

108

Abseitsregel (2)

Graphische Veranschaulichung der Abseitsregel...

```
-----  
kVA r =  
| ((4/3) * myPi * rcube r, 4 * myPi * square r)  
|  
| sphere  
| myPi = 3.14  
| rcube x = x *  
| square x  
|----->  
square x = x * x  
|  
|  
|----->
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

113

Layout-Konventionen

...bewährt hat es sich, eine Layout-Konvention nach folgenden Muster einzuhalten:

```
funlame f1 f2... fn  
| g1 = e1  
| g2 = e2  
| ...  
| gk = ek  
funlame f1 f2... fn  
| dieselsteinanz  
| Besonderstlanger  
Maechter  
= dieselstidindbenso  
| BesonderstlangerAusdruck  
| g2 = e2  
| ...  
| otherwise = ek
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

114

Verantwortung des Programmierers (1)

...die Auswahl einer angemessenen Notation. Vergleiche...

```
triMax :: Integer -> Integer -> Integer  
triMax = \p q r ->  
  if p>=q then (if p>=r then p  
                else r)  
  else (if q>=r then q  
        else r)  
b) triMax p q r =  
  if (p>=q) && (p>=r) then p  
  else  
  if (q>=p) && (q>=r) then q  
  else r  
c) triMax p q r  
  | (p>=q) && (p>=r) = p  
  | (q>=p) && (q>=r) = q  
  | (r>=p) && (r>=q) = r
```

Auswahlkriterium: Welche Variante lässt sich am einfachsten verstehen?

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

115

Verantwortung des Programmierers (2)

Hilfreich ist auch eine Richtschnur von C.A.R. Hoare:

Programme können grundsätzlich auf zwei Arten geschrieben werden:

- So einfach, dass sie offensichtlich keinen Fehler enthalten
- So kompliziert, dass sie keinen offensichtlichen Fehler enthalten

Es liegt am Programmierer, welchen Weg er einschlägt.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

116

Rekursion

...speziell in funktionalen Sprachen

- Das zentrale Ausdrucksmittel/Sprachmittel, Wiederholungen auszudrücken. *Beachte:* Wir haben keine Schleifen in funktionalen Sprachen.
- Erlaubt oft sehr elegante Lösungen, oft wesentlich einfacher als schleifenbasierte Lösungen. Typisches Beispiel: *Türme von Hanoi*.
- Insgesamt so wichtig, dass eine *Klassifizierung* von Rekursionstypen angezeigt ist.

Eine solche Klassifizierung wird uns in der Folge beschäftigen.

Zuvor aber zwei Beispiele: *Quicksort* und *Türme von Hanoi*

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

117

Quicksort

...ein Beispiel, für das Rekursion auf eine elegante Lösung führt:

```
quicksort :: [Int] -> [Int]  
quicksort [] = []  
quicksort (x:xs) =  
  quicksort [ y | y<-xs, y<=x ] ++  
  [x] ++ quicksort [ y | y<-xs, y>x ]
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

118

Türme von Hanoi (1)

...ein anderes Beispiel, für das Rekursion auf eine elegante Lösung führt:

- *Ausgangssituation:* Gegeben sind drei Stapelplätze A, B und C. Auf Platz A liegt ein Stapel unterschiedlich großer Scheiben, die ihrer Größe nach sortiert aufgeschichtet sind, d.h. die Größe der Scheiben nimmt von unten nach oben sukzessive ab.
- *Aufgabe:* Verlege unter Zuhilfenahme von Platz B den Stapel von Scheiben von Platz A auf Platz C, wobei Scheiben stets nur einzeln verlegt werden dürfen und zu keiner Zeit eine größere Scheibe oberhalb einer kleineren Scheibe auf einem der drei Plätze liegen darf.

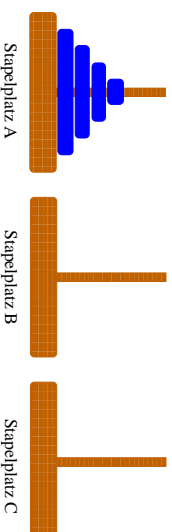
Lösung: Übungsaufgabe

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

119

Türme von Hanoi (2)

Veranschaulichung:

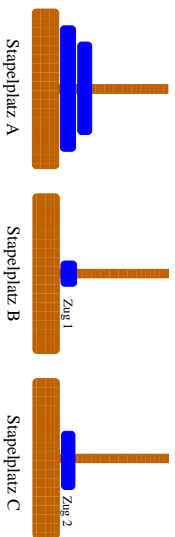


Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

120

Türme von Hanoi (3)

Nach zwei Zügen:

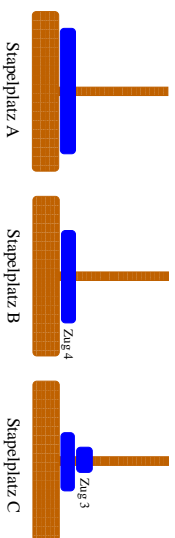


Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

121

Türme von Hanoi (4)

Nach vier Zügen:

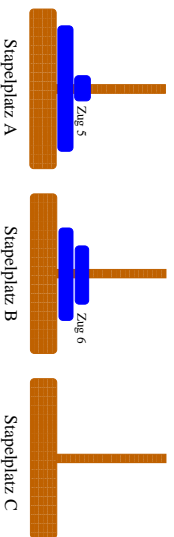


Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

122

Türme von Hanoi (5)

Nach sechs Zügen:

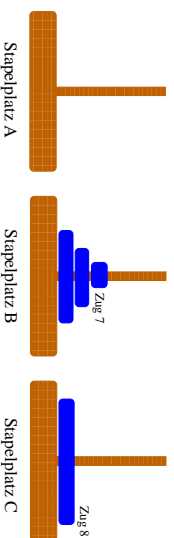


Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

123

Türme von Hanoi (6)

Nach acht Zügen:



Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

124

Klassifikation der Rek.typen (1)

Generell...

...eine Rechenvorschrift heißt *rekursiv*, wenn sie in ihrem Rumpf (direkt oder indirekt) aufgerufen wird.

Dabei können wir unterscheiden...

- *Mikroskopische* Struktur
...betrachtet einzelne Rechenvorschriften und die syntaktische Gestalt der rekursiven Aufrufe
- *Makroskopische* Struktur
...betrachtet Systeme von Rechenvorschriften und ihre gegenseitigen Aufrufe

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

125

Rek.typen: Mikroskopische Struktur (2)

Üblich sind folgende Sprechweisen...

1. Repetitive (schlichte) Rekursion

...pro Zweig höchstens ein rekursiver Aufruf und zwar jeweils als äußerste Operation

Bsp:

```
ggf :: Integer -> Integer -> Integer
ggf m n
  | n == 0           = m
  | m >= n           = ggf (m-1) n
  | m < n             = ggf (n-m) m
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

126

Rek.typen: Mikroskopische Struktur (3)

2. Lineare Rekursion

...pro Zweig höchstens ein rekursiver Aufruf, jedoch nicht notwendig als äußerste Operation

Bsp:

```
powerThree :: Integer -> Integer
powerThree n
  | n == 0 = 1
  | n > 0  = 3 * powerThree (n-1)
```

Beachte: ...im Zweig $n > 0$ ist "*" die äußerste Operation, nicht `powerThree!`

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

127

Rek.typen: Mikroskopische Struktur (4)

3. Geschachtelte Rekursion

...rekursive Aufrufe enthalten rekursive Aufrufe als Argumente

Bsp:

```
fun91 :: Integer -> Integer
fun91 n
  | n > 100 = n - 10
  | n <= 100 = fun91(fun91(n+11))
```

Preisfrage: Warum heißt die Funktion wohl `fun91`?

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

128

Rek.typen: Mikroskopische Struktur (5)

4. Baumartige (kaskadenartige) Rekursion

...pro Zweig können mehrere rekursive Aufrufe nebeneinander vorkommen

Bsp:

```
binom :: (Integer,Integer) -> Integer
binom (n,k)
  | k==0 || n==k  = 1
  | otherwise     = binom (n-1,k-1) + binom (n-1,k)
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

129

Rek.typen: Makroskopische Struktur (6)

1. Direkte Rekursion

...entspricht Rekursion (Präzisierung!)

2. Indirekte oder auch verschränkte (wechselweise) Rekursion

...zwei oder mehr Funktionen rufen sich wechselseitig auf

Bsp:

```
isEven :: Integer -> Bool
isEven n
  | n == 0    = True
  | n > 0    = isOdd (n-1)

isOdd :: Integer -> Bool
isOdd n
  | n == 0    = False
  | n > 0    = isEven (n-1)
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

130

Anm. zu Effektivität & Effizienz (1)

Viele Probleme lassen sich...

- elegant rekursiv lösen (z.B. Türme von Hanoi)
- jedoch nicht immer effizient (\neq effektiv)

Als Faustregel gilt...

- Unter Effizienzgesichtspunkten ist...
 - repetitive Rekursion am (kosten-) günstigsten
 - geschachtelte und baumartige Rekursion am ungünstigsten

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

131

Anm. zu Effektivität & Effizienz (2)

(Oft) folgende Abhilfe bei ineffizienten Implementierungen möglich:

~> Umformulieren! Ersetzen ungünstiger durch günstigere Rekursionsmuster!

Etwas...

- Rückführung linearer Rekursion auf repetitive Rekursion

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

132

Anm. zu Effektivität & Effizienz (3)

...am Beispiel der Fakultätsfunktion:

Naheliegende Formulierung mit *linearer* Rekursionsmuster...

```
fac :: Integer -> Integer
fac n = if n == 0 then 1 else (n * fac(n-1))
```

Effizientere Formulierung mit *repetitivem* Rekursionsmuster...

```
fac :: Integer -> Integer
fac n = facRep (n,1)

facRep :: (Integer,Integer) -> Integer
facRep (p,r) = if p == 0 then r else facRep (p-1,p*r)
```

~> "Trick" ...Rechnen auf Parameterposition!

Aber: Überlagerungen mit anderen Effekten sind möglich, so dass sich der Effizienzgewinn nicht realisiert! (Zur Übung: Wie ist das im obigen Beispiel?)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

134

Fibonacci-Zahlen (1)

Die naheliegende Implementierung...

```
fib :: Integer -> Integer
fib n
  | n == 0  = 0
  | n == 1  = 1
  | otherwise = fib (n-1) + fib (n-2)
```

...führt auf kaskaden- bzw. baumartige Rekursion

~> ... und ist sehr, seeehr laaaaansaaaam (ausprobieren!)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

135

Fibonacci-Zahlen (2)

Veranschaulichung ...durch manuelle Auswertung

fib 0 => 0 -- 1 Aufrufe von fib

fib 1 => 1 -- 1 Aufrufe von fib

fib 2 => fib 1 + fib 0

=> 1 + 0

=> 1 -- 3 Aufrufe von fib

fib 3 => fib 2 + fib 1

=> (fib 1 + fib 0) + 1

=> (1 + 0) + 1

=> 2 -- 5 Aufrufe von fib

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

136

Fibonacci-Zahlen (3)

fib 4 => fib 3 + fib 2
 => (fib 2 + fib 1) + (fib 1 + fib 0)
 => ((fib 1 + fib 0) + 1) + (1 + 0)
 => ((1 + 0) + 1) + (1 + 0)
 => 3 -- 9 Aufrufe von fib

fib 5 => fib 4 + fib 3
 => (fib 3 + fib 2) + (fib 2 + fib 1)
 => ((fib 2 + fib 1) + (fib 1 + fib 0))
 + ((fib 1 + fib 0) + 1)
 => (((fib 1 + fib 0) + 1) + (1 + 0)) + ((1 + 0) + 1)
 => (((1 + 0) + 1) + (1 + 0)) + ((1 + 0) + 1)
 => 5 -- 15 Aufrufe von fib

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

137

Fibonacci-Zahlen (4)

fib 8 => fib 7 + fib 6
 => (fib 6 + fib 5) + (fib 5 + fib 4)
 => ((fib 5 + fib 4) + (fib 4 + fib 3))
 + ((fib 4 + fib 3) + (fib 3 + fib 2))
 => (((fib 4 + fib 3) + (fib 3 + fib 2))
 + (fib 3 + fib 2) + (fib 2 + fib 1))
 + ((fib 3 + fib 2) + (fib 2 + fib 1))
 + ((fib 2 + fib 1) + (fib 1 + fib 0)))
 => ... -- 60 Aufrufe von fib

Offensichtliche Probleme

- viele Mehrfachberechnungen
- exponentielles Wachstum!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

138

Abhilfe

Programmiertechniken wie

- Dynamische Programmierung
- Memoization

Zentrale Idee:

- Speicherung und Wiederverwendung bereits berechneter (Teil-) Ergebnisse statt deren Wiederberechnung.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

139

Komplexitätsklassen (1)

Nach P. Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*, 2. Auflage, 2003, Kapitel 11.

Erinnerung ... O -Notation

- Sei f eine Funktion $f : \alpha \rightarrow \mathbb{R}^+$ von einem gegebenen Datentyp α in die Menge der positiven reellen Zahlen. Dann ist die Klasse $O(f)$ die Menge aller Funktionen, die "langsamer wachsen" als f :

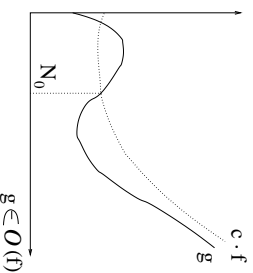
$$O(f) = \{g \mid \exists h(n) \leq \alpha * f(n) \text{ für eine positive Konstante } c \text{ und alle } n \geq N_0\}$$

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

140

Komplexitätsklassen (2)

Veranschaulichung:



Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

141

Komplexitätsklassen (3)

Beispiele häufig auftretender Kostenfunktionen...

Kürzel	Aufwand	Intuition: vertausendfachte Eingabe heißt...
$O(c)$	Konstant	... gleiche Arbeit
$O(\log n)$	logarithmisch	... nur zehnfache Arbeit
$O(n)$	linear	... auch vertausendfache Arbeit
$O(n \log n)$	" $n \log n$ "	... zehntausendfache Arbeit
$O(n^2)$	quadratisch	... millionenfache Arbeit
$O(n^3)$	kubisch	... milliardenfache Arbeit
$O(n^c)$	polynomial	... gigantisch viel Arbeit (für großes c)
$O(2^n)$	exponentiell	... hoffnungslos

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

142

Komplexitätsklassen (4)

...und was wachsende Eingaben in realen Zeiten in der Praxis bedeuten können:

n	linear	quadratisch	kubisch	exponentiell
1	1 μ s	1 μ s	1 μ s	2 μ s
10	10 μ s	100 μ s	1 ms	1 ms
20	20 μ s	400 μ s	8 ms	1 s
30	30 μ s	900 μ s	27 ms	18 min
40	40 μ s	2 ms	64 ms	13 Tage
50	50 μ s	3 ms	125 ms	36 Jahre
60	60 μ s	4 ms	216 ms	36 560 Jahre
100	100 μ s	10 ms	1 sec	4 * 10^{16} Jahre
1000	1 ms	1 sec	17 min	sehr, sehr lange...

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

143

Fazit

Die vorigen Folien machen deutlich...

- ...Effizienz ist wichtig!

- ...Rekursionsmuster haben einen erheblichen Einfluss darauf (vgl. baumartig-rekursive Implementierung der Fibonacci-Funktion. Beachte aber: Nicht das baumartige Rekursionsmuster ist ein Problem an sich, sondern die unnötigen Mehrfachberechnungen von Werten im Falle der Fibonacci-Funktion!)

Allerdings...

- Baumartig rekursive Funktionsdefinitionen bieten sich zur *Parallelisierung* an!
Stichwort: ...divide and conquer!

Zur Übung empfohlen...

- Wie könnte die Berechnung der Folge der Fibonacci-Zahlen

Struktur von Programmen

Programme funktionaler Programmiersprachen, speziell Haskell-Programme, sind zumeist

- Systeme (wechselfeier) *rekursiver* Rechenvorschriften, die sich *hierarchisch* oder/und *wechselfeier* aufeinander abstützen.

Um sich über die *Struktur* solcher Systeme von Rechenvorschriften Klarheit zu verschaffen, ist neben der Untersuchung der beteiligten Rechenvorschriften insbesondere auch die Untersuchung

- ihrer *Aufrufgraphen* geeignet.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 145

Aufrufgraphen

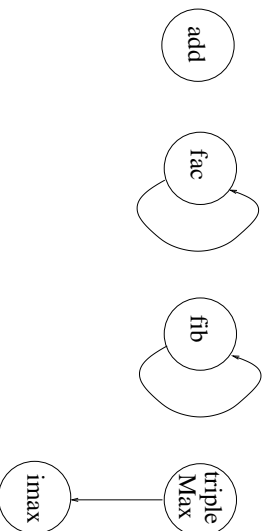
Der *Aufrufgraph* eines Systems *S* von Rechenvorschriften enthält

- einen *Knoten* für jede in *S* deklarierte Rechenvorschrift,
- eine gerichtete *Kante* vom Knoten *f* zum Knoten *g* genau dann, wenn im Rumpf der zu *f* gehörigen Rechenvorschrift die zu *g* gehörige Rechenvorschrift aufgerufen wird.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 146

Beispiele von Aufrufgraphen (1)

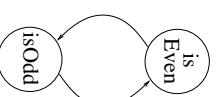
...die Aufrufgraphen des Systems von Rechenvorschriften der Funktionen *add*, *fac*, *fib*, *imax* und *tripleMax*:



Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 147

Beispiele von Aufrufgraphen (2)

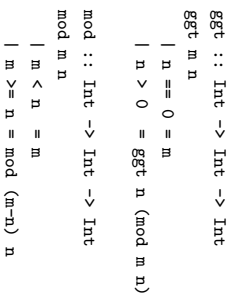
...die Aufrufgraphen des Systems von Rechenvorschriften der Funktionen *isOdd* und *isEven*:



Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 148

Beispiele von Aufrufgraphen (3)

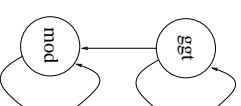
...das System von Rechenvorschriften der Funktionen *ggT* und *mod*:



Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 149

Beispiele von Aufrufgraphen (3)

...und sein Aufrufgraph:



Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 150

Auswertung von Aufrufgraphen

Aus dem Aufrufgraphen eines Systems von Rechenvorschriften ist u.a. ableisbar...

- *Direkte Rekursivität* einer Funktion: "Selbstknoten".
...z.B. bei den Aufrufgraphen der Funktionen *fac* und *fib*.
- *Wechselfeier Rekursivität* zweier (oder mehrerer) Funktionen: Kreise (mit mehr als einer Kante)
...z.B. bei den Aufrufgraphen der Funktionen *isOdd* und *isEven*.
- *Direkte hierarchische Abstützung* einer Funktion auf eine andere: Es gibt eine Kante von Knoten *f* zu Knoten *g*, aber nicht umgekehrt.
...z.B. bei den Aufrufgraphen der Funktionen *tripleMax* und *imax*.
- *Indirekte hierarchische Abstützung* einer Funktion auf eine andere: Knoten *g* ist von Knoten *f* über eine Folge von Kanten erreichbar, aber nicht umgekehrt.

• *Wechselfeier Abstützung*: Knoten *g* ist von Knoten *f* direkt oder indirekt über eine Folge von Kanten erreichbar und umgekehrt.

• *Unabhängigkeit/Isolation* einer Funktion: Knoten *f* hat (ggf. mit Ausnahme eines Selbstknotens) weder ein- noch ausgehende Kanten.
...z.B. bei den Aufrufgraphen der Funktionen *add*, *fac* und *fib*.

• ...

Kapitel 2: Datentypdeklarationen

Datenstrukturen in Haskell...

- Algebraische Datentypen (*data Tree = ...*)
- Typsynonyme (*type Student = ...*)
- Spezialitäten (*newtype State = ...*)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 152

Datentypdeklarationen in Haskell

...selbstdefinierte (neue) Datentypen in Haskell!

~> Haskell's Vehikel dafür: *Algebraische Typen*

Algebraische Typen erlauben uns zu definieren...

- Summentypen
 - *Spezialfälle*
 - * Produkttypen
 - * Aufzählungstypen

In der Praxis besonders wichtige Varianten...

- Rekursive Typen (~> "unendliche" Datenstrukturen)
- Polymorphe Typen (~> Wiederverwendung): *Später!*

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

153

Grundlegende Typmuster

Aufzählungs-, Produkt- und Summentypen:

• *Aufzählungstypen*

~> Typen mit endlich vielen Werten

...:typisches Beispiel: Typ Jahreszeiten mit Werten Frühling, Sommer, Herbst und Winter.

• *Produkttypen* (Synonym: *Verbindtypen*)

~> Typen mit möglicherweise unendlich vielen Tupelwerten

...:typisches Beispiel: Typ Person mit Werten (Adam, männlich, 27), (Eva, weiblich, 25), etc.

• *Summentypen* (Synonym: *Vereinigungstypen*)

~> Vereinigung von Typen mit möglicherweise jeweils unendlich vielen Werten

...:typisches Beispiel: Typ Verkehrsmittel als Vereinigung der (Werte der) Typen Auto, Schiff, Flugzeug, etc.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

154

Zum Einstieg und Vergleich... (1)

Realisierung von Typdefinitionen in imperativen Sprachen

...hier am Bsp. von Pascal

- *Aufzählungstypen*

```
TYPE jahreszeiten = (fruehling, sommer, herbst, winter);
spielkartenfarben = (krenz, pik, herz, karo);
werkzeuge = (montag, dienstag, mittwoch,
             donnerstag, freitag);
transportmittel = (fahrrad, auto, schiff, flugzeug);
form = (kreis, rechteck, quadrat, dreieck);
```

- *Produkttypen*

```
TYPE person = RECORD
    name: ARRAY [1..42] OF char;
    geschlecht: (maennlich, weiblich);
    alter: integer
END;
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

155

Zum Einstieg und Vergleich... (2)

- *Summentypen*

TYPE verkehrsmittel =

```
RECORD
CASE vkm: transportmittel OF
    fahrrad: (tandem: Boolean);
    auto: (hersteller: ARRAY [1..30] OF char;
          hubraum: real);
    schiff: (name: ARRAY [1..30] OF char;
            tiefgang: real;
            heimathafen: ARRAY [1..50] OF char);
    flugzeug: (reichweite: real;
              sitzplaetze: integer)
END;
```

```
geometrischefigur =
RECORD
CASE fgr: form OF
    kreis: (radius: real);
    rechteck: (breite, hoehe: real);
    quadrat: (seitenlaenge, diagonale: real);
    dreieck: (s1, s2, s3: real; rechteckg: boolean);
END;
```

Zum Einstieg und Vergleich... (4)

Obige Einstiegsdatentypbeispiele in Haskell...

- *Aufzählungstyp* Jahreszeiten

```
data Jahreszeiten = Fruehling | Sommer | Herbst | Winter
data Werkzeuge = Montag | Dienstag | Mittwoch | Donnerstag | Freitag
data Bool = True | False
```

- *Produkttyp* Person

```
data Person = Pers Name Geschlecht Alter
```

mit

```
type Name = String
type Alter = Int
data Geschlecht = Maennlich | Weiblich
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

158

Zum Einstieg und Vergleich... (5)

- Summentyp Verkehrsmittel

```
data Verkehrsmittel = Fahrrad Bool |
                    Auto String Float |
                    Schiff String Float String |
                    Flugzeug Float Int
```

In obiger Form offenbar wenig transparent im Vergleich zu:

TYPE verkehrsmittel =

```
RECORD
CASE vkm: transportmittel OF
    fahrrad: (tandem: Boolean);
    auto: (hersteller: ARRAY [1..30] OF char;
          hubraum: real);
    schiff: (name: ARRAY [1..30] OF char;
            tiefgang: real;
            heimathafen: ARRAY [1..50] OF char);
    flugzeug: (reichweite: real;
              sitzplaetze: integer)
END;
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

159

Zum Einstieg und Vergleich... (5)

- Summentyp Verkehrsmittel

```
data Verkehrsmittel = Fahrrad Tandem |
                    Auto Hersteller Hubraum |
                    Schiff Name Tiefgang Heimathafen |
                    Flugzeug Spannweite Sitzplaetze
```

mit

```
type Tandem = Bool
type Hersteller = String
type Hubraum = Float
type Name = String
type Tiefgang = Float
type Heimathafen = String
type Reichweite = Float
type Sitzplaetze = Int
```

Man erkennt: Typsynonyme bringen Transparenz ins Programm!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

160

Algebraische Datentypen in Haskell

...das allg. Muster der algebraischen Datentypdefinition:

```
data Typename
= Con1 t11 ... t1k1 |
  Con2 t21 ... t2k2 |
  ...
  Conn tn1 ... tnkn
```

Sprechweisen:

- Typename ... *Typname/-identifikator*
- $\text{Con}_i, i = 1..n$... *Konstruktor(en)/-identifikatoren*
- $k_i, i = 1..n$... *Stelligkeit des Konstruktors* $\text{con}_i, k_i \geq 0$,
 $i = 1, \dots, n$

Beachte: Typ- und Konstruktorenidentifikatoren müssen mit einem Großbuchstaben beginnen (siehe z.B. True, False)!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

162

Konstruktoren...

...können als Funktionsdefinitionen gelesen werden:

```
Coni :: t11 -> ... -> t1ki -> Typename
```

Konstruktion von Werten eines algebraischen Datentyps durch...

...Anwendung eines Konstruktors auf Werte "passenden" Typs, d.h.

```
Coni v11 ... v1ki :: Typname  
wobei v11 :: t11 ... v1ki :: t1ki, j = 1, ..., ki
```

Beispiele:

- Pers "Adam" Maennlich 27 :: Person
- Schiff "Donaukönigin" 2.74 "Wien" :: Verkehrsmittel
- Flugzeug 8540.75 275 :: Verkehrsmittel

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

162

Algebraische Datentypen in Haskell

...das allg. Muster der algebraischen Datentypdefinition:

```
data Typename
= Con1 t11 ... t1k1 |
  Con2 t21 ... t2k2 |
  ...
  Conn tn1 ... tnkn
```

Sprechweisen:

- Typename ... *Typname/-identifikator*
- $\text{Con}_i, i = 1..n$... *Konstruktor(en)/-identifikatoren*
- $k_i, i = 1..n$... *Stelligkeit des Konstruktors* $\text{con}_i, k_i \geq 0$,
 $i = 1, \dots, n$

Beachte: Typ- und Konstruktorenidentifikatoren müssen mit einem Großbuchstaben beginnen (siehe z.B. True, False)!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

162

Aufzählungstypen (1)

Nullstellige Konstruktoren führen auf *Aufzählungstypen*...

Beispiele:

```
data Spielfarbe = Kreuz | Pik | Herz | Karo
data Wochenende = Sonnabend | Sonntag
data Geschlecht = Maennlich | Weiblich
data Form       = Kreis | Rechteck | Quadrat | Dreieck
```

Insbesondere ist der Typ der Wahrheitswerte...

```
data Bool = True | False
```

...Beispiel eines in Haskell vordefinierten Aufzählungstyps.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

163

Aufzählungstypen (2)

Funktionsdefinitionen über Aufzählungstypen...

~> üblicherweise mit Hilfe von Pattern-matching.

Beispiele:

```
hateEcken :: Form -> Bool
hateEcken Kreis = False
hateEcken _     = True
```

```
istLandegebunden :: Verkehrsmittel -> Bool
istLandegebunden Fahrrad = True
istLandegebunden Auto    = True
istLandegebunden _       = False
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

164

Produkttypen

(Alternativenlose) mehrstellige Konstruktoren führen auf *Produkttypen*...

Beispiel:

```
type Name      = String
type Alter     = Int
data Geschlecht = Maennlich | Weiblich
```

```
data Person = Pers Name Geschlecht Alter
```

Beispiele: ...für Werte des Typs Person.

```
Pers "Paul Pfiffig" Maennlich 23 :: Person
Pers "Paula Pfietsch" Weiblich 22 :: Person
```

Beachte: Funktionalität der Konstruktorfunktion ist hier...

```
Pers :: Name -> Geschlecht -> Alter -> Person
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

165

Summentypen (1)

Mehrere (null- oder mehrstellige) Konstruktoren führen auf *Summentypen*...

Beispiel:

```
type Radius   = Float
type Breite   = Float
type Hoehe    = Float
type Seite1   = Float
type Seite2   = Float
type Seite3   = Float
type Rechtwinklig = Bool
```

```
data XFigur = Kreis Radius |
            Rechteck Breite Hoehe |
            Quadrat Kantenlaenge |
            Dreieck Seite1 Seite2 Seite3 Rechtwinklig |
            Ebene
```

Die Varianten einer Summe werden durch "|" getrennt.

165

Summentypen (2)

Beispiele: ...für Werte des Typs erweiterte Figur XFigur

```
Kreis 3.14      :: XFigur
Rechteck 17.0 4.0 :: XFigur
Quadrat 47.11   :: XFigur
Dreieck 3.0 4.0 5.0 True :: XFigur
Ebene      :: XFigur
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

167

Zwischenfazit

Somit ergibt sich die eingangs genannte Taxonomie algebraischer Datentypen...

Haskell offeriert...

• Summentypen

mit den beiden *Spezialfällen*

• Produkttypen

~> nur ein Konstruktor, mehrstellig

• Aufzählungstypen

~> ein oder mehrere Konstruktoren, alle nullstellig

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

168

Rekursive Typen (1)

...der Schlüssel zu (potentiell) unendlichen Datenstrukturen.

Technisch:

... zu definierende Typnamen können rechtsseitig in der Definition benutzt werden.

Beispiel: ... (arithmetische) Ausdrücke

```
data Expr = Opd Int |
  Add Expr Expr |
  Sub Expr Expr |
  Squ Expr
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

169

Rekursive Typen (2)

Beispiele ... für Ausdrücke (lies --> als "entspricht").

```
Opd 42 :: Expr          --> 42
Add (Opd 17) (Opd 4) :: Expr      --> 17+4
Add (Squ (Sub (Opd 42) (Squ (2)))) (Opd 12) :: Expr
--> square(42-square(2))+12
```

...rekursive Typen ermöglichen potentiell unendliche Datenstrukturen!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

170

Rekursive Typen (3)

Weiteres Beispiel:

Binärbäume, hier zwei verschiedene Varianten:

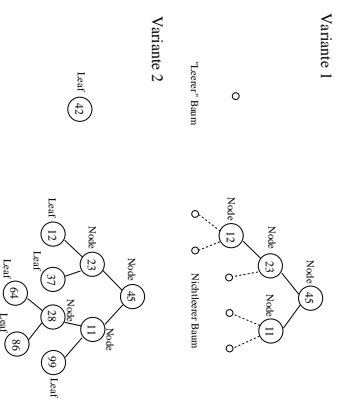
```
data BinTree1 = Nil | Node Int BinTree1 BinTree1
data BinTree2 = Leaf Int | Node Int BinTree2 BinTree2
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

171

Rekursive Typen (4)

Veranschaulichung der Binärbaumvarianten 1&2 anhand eines Beispiels:



Rekursive Typen (5)

Beispiele ... für Funktionen über Binärbaumvariante 1.

```
valBinTree1 :: BinTree1 -> Int
valBinTree1 Nil           = 0
valBinTree1 (Node n b1 b2) = n + valBinTree1 b1 + valBinTree1 b2

dephBinTree1 :: BinTree1 -> Int
dephBinTree1 Nil = 0
dephBinTree1 (Node _ b1 b2)
  = 1 + max (dephBinTree1 b1) (dephBinTree1 b2)
```

Mit diesen Definitionen...

```
valBinTree1 Nil           == 0
valBinTree1 (Node 17 Nil (Node 4 Nil Nil)) == 21
dephBinTree1 (Node 17 Nil (Node 4 Nil Nil)) == 2
dephBinTree1 Nil         == 0
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

173

Rekursive Typen (6)

Beispiele ... für Funktionen über Binärbaumvariante 2.

```
valBinTree2 :: BinTree2 -> Int
valBinTree2 (Leaf n)           = n
valBinTree2 (Node n b1 b2)    = n + valBinTree2 b1 + valBinTree2 b2

dephBinTree2 :: BinTree2 -> Int
dephBinTree2 (Leaf _) = 1
dephBinTree2 (Node _ b1 b2)
  = 1 + max (dephBinTree2 b1) (dephBinTree2 b2)
```

Mit diesen Definitionen...

```
valBinTree2 (Leaf 3)           == 3
valBinTree2 (Node 17 (Leaf 4) (Node 4 (Leaf 12) (Leaf 5))) == 42
dephBinTree2 (Node 17 (Leaf 4) (Node 4 (Leaf 12) (Leaf 5))) == 3
dephBinTree2 (Leaf 3)         == 1
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

174

Wechselweise rekursive Typen

...ein Spezialfall rekursiver Typen.

Beispiel:

```
data Individual = Adult Name Address Biography |
  Child Name

data Biography = Parent String [Individual] |
  NonParent String
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

175

Typsynonyme (1)

...hatten wir bereits kennengelernt bei der Einführung von Tupeltypen:

```
type Student = (String, String, Int)
type Buch = (String, String, Int, Bool)
```

...und auch in den Beispielen zu algebraischen Datentypen benutzt:

```
data Verkehrsmitel = Fahrrad Tandem |
  Auto Herkoller Hubraum |
  Schiff Name Tiefgang Heimathafen |
  Flugzeug Spannweite Sitzplatze

type Tandem      = Bool
type Herkoller  = String
type Hubraum     = Float
type Name       = String
type Tiefgang   = Float
type Heimathafen = String
type Reichweite = Float
type Sitzplatze = Int
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

176

Typsynonyme (2)

- Das Schlüsselwort `type` leitet die Deklaration von Typsynonymen ein
- Unbedingt zu beachten ist...
 - `type` ..führt neue Namen für bereits existierende Typen ein (Typsynonymel). keine neuen Typen.

Somit gilt:

Durch `type`-Deklarationen eingeführte Typsynonyme...

- tragen zur Dokumentation bei und
- erleichtern (bei treffender Namenswahl) das Programmverständnis
- führen nicht zu (zusätzlicher) Typsicherheit!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 178

Ein (pathologisches) Beispiel

```
type Euro      = Float
type Yen      = Float
type Temperatur = Float

mypi :: Float
daunen :: Float
maxTemp :: Temperature
mypi_1 = 3.14
daunen = 5.55
maxTemp = 43.2
```

```
currencyConverter :: Euro -> Yen
currencyConverter x = x + myPi * daunen
```

Mit obigen Deklarationen...

```
currencyConverter maxTemp => 60.627
```

....werden 43.2 °C in 60.627 Yen umgerechnet. Typsicher?

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 178

Ein reales Beispiel

Anflugsteuerung einer Sonde zum Mars...

```
type Geschwindigkeit = Float
type Weilen          = Float
type Km              = Float
type Zeit            = Float
type Wegstrecke     = Meilen
type Distanz        = Km
```

```
geschwindigkeit :: Wegstrecke -> Zeit -> Geschwindigkeit
geschwindigkeit w z = (/) w z
```

```
verbleibendeFlugzeit :: Distanz -> Geschwindigkeit -> Zeit
verbleibendeFlugzeit d g = (/) d g
```

```
verbleibendeFlugzeit 18524.34 1523.79
```

...durch Typisierungsprobleme dieser Art ging vor einigen Jahren eine Marssonde im Wert von mehreren 100 Mill. USD verloren.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 179

Produkttypen vs. Tupeltypen (1)

Der Typ `Person` als...

- *Produkttyp*

```
data Person = Pers Name Geschlecht Alter
```
- *Tupeltyp*

```
type Person = (Name, Geschlecht, Alter)
```

Vordergründiger Unterschied:

...in der Tupeltypvariante fehlt der Konstruktor (in diesem Bsp.: `Pers`)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 180

Produkttypen vs. Tupeltypen (2)

...eine Abwägung von Vor- und Nachteilen.

Produkttypen und ihre typischen...

- *Vorteile* gegenüber *Tupeltypen*
 - Objekte des Typs sind mit dem Konstruktor "markiert" (trägt zur Dokumentation bei)
 - Tupel mit zufällig passenden Komponenten nicht irrtümlich als Elemente des Produkttyps manipulierbar (Typsicherheit! Vgl. früheres Beispiel zur Umrechnung Euro in Yen!)
 - Aussagekräftigere (Typ-) Fehlermeldungen (Typsynonyme können wg. Expansion in Fehlermeldungen fehlen).
- *Nachteile* gegenüber *Tupeltypen*
 - Produkttypenelemente sind weniger kompakt, erfordern längere Definitionen (mehr Schreibarbeit)
 - Auf Tupeln vordefinierte polymorphe Funktionen (z.B. `fst`, `snd`, `zip`, `unzip`, ...) stehen nicht zur Verfügung.
 - Der Code ist weniger effizient.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 181

Andererseits...

Mit Produkttypen statt Typsynonymen...

```
data Euro      = EUR Float
data Yen      = YEN Float
data Temperatur = Temp Float
```

```
mypi :: Float
daunen :: Float
maxTemp :: Temperature
mypi_1 = 3.14
daunen = 5.55
maxTemp = Temp 43.2
```

...wäre eine Funktionsdefinition im Stile von `currencyConverter :: Euro -> Yen`
`currencyConverter x = x + myPi * daunen`
insbesondere auch ein Aufruf wie...

```
currencyConverter maxTemp
```

durch das Typsystem von Haskell verhindert!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 182

Somit als kurzes Fazit... (1)

...unserer Überlegungen:

- *Typsynonyme* wie...

```
type Euro      = Float
type Yen      = Float
type Temperatur = Float
```

...erben alle Operationen von `Float` und sind damit beliebig austauschbar – mit allen Annehmlichkeiten und Gefahren, sprich Fehlerquellen.

- *Produkttypen* wie...

```
data Euro      = EUR Float
data Yen      = YEN Float
data Temperatur = Temp Float
```

...erben keinerlei Operationen von `Float`, bieten dafür aber um den Preis zusätzlicher Schreibarbeit und gewissen Performanceverlusts Typsicherheit!

Somit als kurzes Fazit... (2)

In ähnlicher Weise...

```
data Miles      = Mi Float
data Km         = Km Float
type Distance   = Miles
type Wegstrecke = Km
...
```

...wäre auch der Verlust der Marssonde vermutlich vermeidbar gewesen.

Beachte:

- Typ- und Konstruktornamen dürfen in Haskell übereinstimmen (siehe z.B. `data Km = Km Float`)
- Konstruktornamen müssen global (d.h. modulweise) eindeutig sein.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 184

Spezialitäten

...die `newtype`-Deklaration:

```
newtype Miles = Mi Float
```

`newtype`-Deklarationen sind im Hinblick auf...

- Typsicherheit
- ...mit `data`-Deklarationen vergleichbar
- Effizienz
- ...mit `type`-Deklarationen vergleichbar

Beachte: `newtype`-Deklarationen sind auf Typen mit nur einem Konstruktor eingeschränkt.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

185

Polymorphe Typen

...in der Folge!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

186

Rückblick: Das zentrale Thema zuletzt...

...selbstdefinierte (neue) Datentypen in Haskell!

~ Haskell's Vehikel dafür: *Algebraische Typen*

Algebraische Typen erlauben uns zu definieren...

- Summentypen
 - Spezialfälle
 - * Produkttypen
 - * Aufzählungstypen

In der Praxis besonders wichtige Varianten...

- Rekursive Typen (~ "unendliche" Datenstrukturen)
- Polymorphe Typen (~ Wiederverwendung)

Offen geblieben ist bislang die Untersuchung und Diskussion *polymorpher Typen!*

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

187

Kapitel 3: Polymorphe Funktionen und Datentypen

Polymorphie

- Bedeutung lt. Duden:

– *Vielseitigkeit, Verschiedengestaltigkeit*

...mit speziellen fachspezifischen Bedeutungsausprägungen

* In der *Chemie*: das Vorkommen mancher Mineralien in verschiedener Form, mit verschiedenen Eigenschaften, aber gleicher chemischer Zusammensetzung

* In der *Biologie*: Vielseitigkeit der Blätter oder der Blüte einer Pflanze

* In der *Sprachwissenschaft*: das Vorhandensein mehrerer sprachlicher Formen für den gleichen Inhalt, die gleiche Funktion (z.B. die verschiedenartigen Pluralbildungen in: die Wiesen, die Felder, die Tiere)

* In der *Informatik*, speziell der *Theorie der Programmiersprachen*:
~ das nächste Thema!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

188

Polymorphie

...im programmiersprachlichen Kontext unterscheiden wir insbesondere zwischen

- Polymorphie
 - auf Datentypen
 - auf Funktionen
 - * Parametrische Polymorphie ("Echte Polymorphie")
 - * Ad hoc Polymorphie (synonym: Überladung)
 - ~ Haskell-spezifisch: Typklassen

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

189

Polymorphie

Wir beginnen mit...

- (Parametrischer) Polymorphie auf Funktionen
- ...die wir an einigen Beispielen schon kennengelernt haben:
- Die Funktionale `curry` und `uncurry`
- Die Funktionen `length`, `head` und `tail`

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

190

Rückblick (1)

Die Funktionale `curry` und `uncurry`...

```
curry :: ((a,b) -> c) -> (a -> b -> c)
```

```
curry f x y = f (x,y)
```

```
uncurry :: (a -> b -> c) -> ((a,b) -> c)
```

```
uncurry g (x,y) = g x y
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

191

Rückblick (2)

Die Funktionen `length`, `head` und `tail`...

```
length :: [a] -> Int
```

```
length [] = 0
```

```
length (_:xs) = 1 + length xs
```

```
head :: [a] -> a
```

```
head (x:_) = x
```

```
tail :: [a] -> [a]
```

```
tail (.:xs) = xs
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

192

Äußeres Kennzeichen parametrischer Polymorphie

Statt

- (ausschließlich) konkreter Typen (wie Int, Bool, Char,...) treten in der (Typ-) Signatur der Funktionen
- (auch) *Typparameter*, sog. *Typvariablen* auf.

Beispiele:

```
curry :: ((a,b) -> c) -> (a -> b -> c)
```

```
length :: [a] -> Int
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

193

Typvariablen in Haskell

Typvariablen in Haskell sind...

- freigeählte Identifikatoren, die mit einem Kleinbuchstaben beginnen müssen
- z.B.: a, b, aber auch fp185161

Im Unterschied dazu sind Typnamen, (Typ-) Konstruktoren in Haskell...

- freigeählte Identifikatoren, die mit einem Großbuchstaben beginnen müssen
- z.B.: A, String, Node, aber auch Fp185161_M50708

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

194

Warum Polymorphie auf Funktionen?

...Wiederverwendung (durch Abstraktion)!

~> *ein typisches Vorgehen in der Informatik!*

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

195

Einfaches Bsp.: Funktionsabstraktion

Sind viele Ausdrücke der Art

```
(5 * 37 + 13) * (37 + 5 * 13)
(15 * 7 + 12) * (7 + 15 * 12)
(25 * 3 + 10) * (3 + 25 * 10)
...
```

zu berechnen, schreibe eine Funktion

```
f :: Int -> Int -> Int -> Int
f a b c = (a * b + c) * (b + a * c)
```

um die Rechenvorschrift (a * b + c) * (b + a * c) wiederverwenden zu können:

```
f 5 37 13
f 15 7 12
f 25 3 10
...
```

~> sog. *Funktionale Abstraktion* (verwandt: *Prozedurale Abstraktion*)

Zur Motivation param. Polymorphie (1)

Listen können Elemente sehr unterschiedlicher Typen zusammenfassen, z.B.

- Listen von Basistypen
[2,4,23,2,53,4] :: [Int]
- Listen von Listen
[[2,4,23,2,5], [3,4], [], [56,7,6,1]] :: [[Int]]
- Listen von Paaren
[(3,14,42.0), (56,1,51.3), (1,12,2.22)] :: [Point]
- Listen von Bäumen
[Nil,Node 42 Nil Nil], Node 17 (Node 4 Nil Nil) Nil]
- ...
- Listen von Funktionen
[fact, fib, fun91] :: [Integer -> Integer]

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

198

Zur Motivation param. Polymorphie (2)

- *Aufgabe:* Bestimme die Länge einer Liste, d.h. die Anzahl ihrer Elemente.
- *Naive Lösung:* Schreibe für jeden Typ eine entsprechende Funktion.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

198

Zur Motivation param. Polymorphie (4)

Umsetzung der naiven Lösung:

```
lengthInList :: [Int] -> Int
lengthInList [] = 0
lengthInList (_,xs) = 1 + lengthInList xs

lengthInList :: [[Int]] -> Int
lengthInList [] = 0
lengthInList (_,xs) = 1 + lengthInList xs

lengthPointList :: [Point] -> Int
lengthPointList [] = 0
lengthPointList (_,xs) = 1 + lengthPointList xs

lengthTreeList :: [BinTree] -> Int
lengthTreeList [] = 0
lengthTreeList (_,xs) = 1 + lengthTreeList xs

lengthFunList :: [Integer -> Integer] -> Int
lengthFunList [] = 0
lengthFunList (_,xs) = 1 + lengthFunList xs
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

199

Zur Motivation param. Polymorphie (5)

Damit möglich:

```
lengthInList [2,4,23,2,53,4] => 6
lengthInList [2,4,23,2,5], [3,4], [] , [56,7,6,1] => 4
lengthPointList [(3,14,42.0), (56,1,51.3), (1,12,2.22)] => 3
lengthTreeList [Nil,Node 42 Nil Nil, Node 17 (Node 4 Nil Nil) Nil] => 3
lengthFunList [fact, fib, fun91] => 3
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

200

Zur Motivation param. Polymorphie (6)

Beobachtung:

- Die einzelnen Rechenvorschriften zur Längenberechnung sind i.w. identisch
- Unterschiede beschränken sich auf
 - Funktionsnamen und
 - Typsignaturen

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

201

Zur Motivation param. Polymorphie (7)

Sprachen, die parametrische Polymorphie offerieren, erlauben eine elegantere Lösung unserer Aufgabe:

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs

length [2,4,23,2,53,4] => 6
length [12,4,23,2,5],[13,4],[],[56,7,6,1] => 4
length [(3,14,42,0),(56,1,51,3),(1,12,2,22)] => 3
length [Nil,Node 42 Nil Nil, Node 17 (Node 4 Nil Nil) Nil] => 3
length [fact, fib, fib9] => 3
```

Funktionale Sprachen, auch Haskell, offerieren parametrische Polymorphie!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

202

Zur Motivation param. Polymorphie (8)

Unmittelbare Vorteile parametrischer Polymorphie:

- Wiederverwendung von
 - Rechenvorschriften und
 - Funktionsnamen (*Gute Namen sind knapp!*)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

203

Sprechweisen im Zshg. mit parametrischer Polymorphie

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

Sprechweisen:

- a In der Typsignature von `length` heißt *Typvariable*. Typvariablen werden mit Kleinbuchstaben gewöhnlich vom Anfang des Alphabets bezeichnet: a, b, c,....
- Typen der Form...

```
length :: [Point] -> Int
length :: [[Int]] -> Int
length :: [Integer -> Integer] -> Int
...
```

heißten *Instanzen* des Typs [a] -> Int. Letzterer heißt *allgemeinster Typ* der Funktion `length`.

Bem.: Das Hugs-Kommando `:t expr` liefert stets den (eindeutig bestimmten) *allgemeinsten Typ* eines (wohlgeformten) Haskell-Ausdrucks *expr*.

Weitere in Haskell auf Listen vordefinierte (polymorphe) Funktionen

```
: :: a -> [a] -> [a]
!! :: [a] -> Int -> a
length :: [a] -> Int
++ :: [a] -> [a] -> [a]
concat :: [[a]] -> [a]
head :: [a] -> a
last :: [a] -> a
tail :: [a] -> [a]
init :: [a] -> [a]
splitAt :: Int -> [a] -> ([a],[a])
reverse :: [a] -> [a]
```

Listenkonstruktor (rechtssassoziativ)
Proj. auf *i*-te Komp., `Inf!xop`.
Länge der Liste
Konkatenation zweier Listen
Konkatenation mehrerer Listen
Listenkopf
Listenelement
Liste ohne Listenkopf
Liste ohne Listenelement
Aufspalten einer Liste an Stelle `i`
Umdrehen einer Liste

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

207

Monomorphie vs. Polymorphie

Rechenvorschriften der Form

- `length :: [a] -> Int` heißen *polymorph*.

Rechenvorschriften der Form

- `lengthIntList :: [Int] -> Int`
- `lengthIntList :: [[Int]] -> Int`
- `lengthPointList :: [Point] -> Int`
- `lengthFunList :: [Integer -> Integer] -> Int`
- `lengthTreeList :: [BinTree] -> Int` heißen *monomorph*

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

204

Weitere Bsp. polymorpher Funktionen

```
id :: a -> a
id x = x
-- Identitätsfunktion

id 3 => 3
id ["abc","def"] => ["abc","def"]

zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip [] [] = []
-- "Verschmelzen" von Listen

zip [(3,4,5) ['a','b','c'],'d'] => [(3,'a'),(4,'b'),(5,'c')]
zip ["abc","def","geh"] [(3,4),(5,4)] => [(("abc", (3,4)),("def", (5,4)))]

unzip :: [(a,b)] -> ([a],[b])
-- "Aufreißen" von Listen
unzip [] = ([],[])
unzip [(x,y):(ps) = (x:xs,y:ys)
where
  (xs,ys) = unzip ps

unzip [(3,'a'),(4,'b'),(5,'c')] => ([3,4,5],[('a','b','c')])
unzip [(("abc", (3,4)),("def", (5,4)))] => ([("abc","def"),[(3,4),(5,4)])]
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

205

Polymorphie

Sowiel zu parametrischer Polymorphie auf Funktionen...

Wir fahren fort mit

- Polymorphie auf Datentypen
 - Algebraische Datentypen
 - Typsynonymen

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

208

Polymorphe algebraische Typen (1)

Der Schlüssel dazu...

~> Definitionen algebraischer Typen dürfen Typvariablen enthalten und werden dadurch polymorph.

Beispiele: Paare und Bäume...

```
data Pairs a = Pair a a
```

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

209

Polymorphe algebraische Typen (2)

Beispiele konkreter Werte von Paaren und Bäumen:

```
data Pairs a = Pair a a
```

```
Pair 17 4      :: Pairs Int
```

```
Pair [] [42]  :: Pairs [Int]
```

```
Pair [] []    :: Pairs [a]
```

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

```
Node 'a' (Node 'p' Nil Nil) (Node 'z' Nil Nil)      :: Tree Char
```

```
Node 3.14 (Node 2.0 Nil Nil) (Node 1.41 Nil Nil)    :: Tree Float
```

```
Node "abc" (Node "b" Nil Nil) (Node "xyzzz" Nil Nil) :: Tree [Char]
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

210

Polymorphe algebraische Typen (3)

Ähnlich wie parametrische Polymorphie unterstützt auch...

- Polymorphie auf algebraischen Datentypen

Wiederverwendung!

Vergleiche dies mit der schon bekannten Situation im Zshfg. mit *polymorphen Listen*:

```
length :: [a] -> Int
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

211

Polymorphe Typen (4)

Ähnlich wie bei der Funktion *Länge* auf *Listen*...

```
length :: [a] -> Int
```

```
length [] = 0
```

```
length (x:xs) = 1 + length xs
```

...kann auf algebraischen Typen Typunabhängigkeit generell vorteilhaft ausgenutzt werden, wie hier das Bsp. der Funktion *depth* auf Bäumen zeigt:

```
depth :: Tree a -> Int
```

```
depth Nil = 0
```

```
depth (Node _ t1 t2) = 1 + max (depth t1) (depth t2)
```

```
depth (Node 'a' (Node 'p' Nil Nil) (Node 'z' Nil Nil))    => 2
```

```
depth (Node 3.14 (Node 2.0 Nil Nil) (Node 1.41 Nil Nil)) => 2
```

```
depth (Node "abc" (Node "b" Nil Nil) (Node "xyzzz" Nil Nil)) => 2
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

212

Heterogene algebraische Typen

...sind möglich, z.B. *heterogene* Bäume:

```
data HTree = LeafS String |
           | LeafI Int |
           | NodeF Float HTree HTree |
           | NodeB Bool HTree HTree
```

```
-- 2 Varianten der Funktion Tiefe auf Werten vom Typ HTree
```

```
depth :: HTree -> Int
```

```
depth (LeafS _) = 1
```

```
depth (LeafI _) = 1
```

```
depth (NodeF _ t1 t2) = 1 + max (depth t1) (depth t2)
```

```
depth (NodeB _ t1 t2) = 1 + max (depth t1) (depth t2)
```

```
depth :: HTree -> Int
```

```
depth (NodeF _ t1 t2) = 1 + max (depth t1) (depth t2)
```

```
depth (NodeB _ t1 t2) = 1 + max (depth t1) (depth t2)
```

```
depth _ = 1
```

Beachte: ...folgendes geht nicht ~> *Syntaxfehler!*

```
depth :: HTree -> Int
```

```
depth (_ _ t1 t2) = 1 + max (depth t1) (depth t2)
```

```
depth _ = 1
```

Heterogene polymorphe algeb. Typen

...sind ebenfalls möglich, z.B. *heterogene* polymorphe Bäume:

```
data PHTree a b c d = LeafA a |
                   | LeafB b |
                   | NodeC c (PHTree a b c d) (PHTree a b c d) |
                   | NodeD d c (PHTree a b c d) (PHTree a b c d)
```

```
-- 2 Varianten der Funktion Tiefe auf Werten vom Typ PHTree
```

```
depth :: (PHTree a b c d) -> Int
```

```
depth (LeafA _) = 1
```

```
depth (LeafB _) = 1
```

```
depth (NodeC _ t1 t2) = 1 + max (depth t1) (depth t2)
```

```
depth (NodeD _ _ t1 t2) = 1 + max (depth t1) (depth t2)
```

```
depth :: (PHTree a b c d) -> Int
```

```
depth (NodeC _ t1 t2) = 1 + max (depth t1) (depth t2)
```

```
depth (NodeD _ _ t1 t2) = 1 + max (depth t1) (depth t2)
```

```
depth _ = 1
```

Das heißt...

- Auch "mehrfach" *polymorphe* Datenstrukturen sind möglich!

Polymorphe Typsynonyme

...auch *polymorphe* *Typsynonyme* und *Funktionen* darauf sind möglich:

Beispiel:

```
type List a = [a]
```

```
lengthList :: List a -> Int
```

```
lengthList [] = 0
```

```
lengthList (_:xs) = 1 + lengthList xs
```

Oder kürzer:

```
lengthList :: List a -> Int
```

```
lengthList = length
```

...abstützen auf Standardfunktion *length* möglich, da *List a* Typsynonym, kein neuer Typ.

Beachte: *type List = [a]* ist nicht möglich (~> *Typfehler!*)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

215

Zusammenfassung und erste Schlussfolgerungen

Zu...

- Polymorphen Funktionen,
- Polymorphen Datentypen und
- Vorteilen, die aus ihrer Verwendung resultieren.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

216

Polymorphe Typen

... ein (Daten-) Typ *T* heißt *polymorph*, wenn bei der Deklaration von *T* der Grundtyp oder die Grundtypen der Elemente (in Form einer oder mehrerer Typvariablen) als Parameter angegeben werden.

Zwei Beispiele:

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

data List a = Empty |
            (Head a) (List a)
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

217

Warum polymorphe Typen und Funktionen? (1)

...Wiederverwendung durch Parametrisierung!

~> *wie schon gesagt, ein typisches Vorgehen in der Informatik!*

...die Essenz eines Datentyps (einer Datenstruktur) ist wie die Essenz darauf arbeitender Funktionen oft unabhängig von bestimmten typspezifischen Details.

(Man vergewenwärtige sich das noch einmal z.B. anhand von Bäumen über ganzen Zahlen, Zeichenreihen, Personen,... oder Listen über Gleitkommazahlen, Wahrheitswerten, Bäumen,... und typischen Funktionen darauf wie etwa zur Bestimmung der Tiefe von Bäumen oder der Länge von Listen.)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

219

Warum polymorphe Typen und Funktionen? (3)

Polymorphie und die mit ihr verbundene Wiederverwendung unterstützt die...

- Ökonomie der Programmierung (vulgo: "Schreibfaulheit")

Insbesondere aber trägt sie bei zu höherer...

- Transparenz und Lesbarkeit
...durch Betonung der Gemeinsamkeiten, nicht der Unterschiede!
- Verlässlichkeit und Wartbarkeit (ein Aspekt mit mehreren Dimensionen: Fehlersuche, Weiterentwicklung,...)
...als erwünschte Nebeneffekte!
- ...
- Effizienz (der Programmierung)
~> *höhere Produktivität, früherer Markteintritt (time-to-market)*

Ad hoc Polymorphie

Bisher haben wir besprochen...

- Polymorphie in Form von...
 - (Parametrischer) Polymorphie
 - Polymorphie auf Datentypen

Jetzt ergänzen wir diese Betrachtung um...

- *Ad hoc* Polymorphie (Überladen, "unechte" Polymorphie)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

223

Polymorphe Funktionen

...eine Funktion *f* heißt *polymorph*, wenn deren Parameter (in Form einer oder mehrerer Typvariablen) für Argumente unterschiedlicher Typen definiert sind.

Typische Beispiele:

```
depth :: (Tree a b) -> Int
depth Leaf _ = 1
depth (Node _ t1 t2) = 1 + max (depth t1) (depth t2)

lengthList :: List a -> Int
lengthList Empty = 0
lengthList (Head _ hs) = 1 + lengthList hs

-- Zum Vergleich die polymorphe Standardfunktion length
length :: [a] -> Int
length [] = 0
length (_,:xs) = 1 + length xs
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

218

Warum polymorphe Typen und Funktionen? (2)

• ...durch Parametrisierung werden gleiche Teile "ausgeklammert" und somit der Wiederverwendung zugänglich!

• ...(!w.) gleiche Codeteile müssen nicht (länger) mehrfach geschrieben werden.

• ...man spricht deshalb auch von *parametrischer Polymorphie*.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

220

Warum polymorphe Typen und Funktionen? (4)

Beachte...

• ...auch in anderen Paradigmen wie etwa imperativer und speziell objektorientierter Programmierung lernt man, den Nutzen und die Vorteile polymorpher Konzepte zunehmend zu schätzen!

~> aktuelles Stichwort: *Generic Java*

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

222

Zur Motivation von Ad hoc Polymorphie

Ausdrücke der Form

```
(+) 2 3      => 5
(+) 27.55 12.8 => 39.63
(+) 12.42 3   => 15.42
```

...sind Beispiele wohlgeformter Haskell-Ausdrücke, wohingegen

```
(+) True False
(+) 'a' 'b'
(+) [1,2,3] [4,5,6]
```

...Beispiele nicht wohlgeformter Haskell-Ausdrücke sind.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

224

Zur Motivation von Ad hoc Polymorphie

Offenbar...

- ist (+) nicht monomorph
...da (+) für mehr als einen Argumenttyp arbeitet
- ist der Typ von (+) verschieden von $a \rightarrow a \rightarrow a$
...da (+) nicht für jeden Argumenttyp arbeitet

Tatsächlich...

- ist (+) typisches Beispiel eines *überladenen* Operators.

Das Kommando `:t (+)` in Hugs liefert

- (+) :: Num a => a -> a -> a

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

225

Polymorphie vs. Ad hoc Polymorphie

Intuitiv

- Polymorphie
Der polymorphe Typ $(a \rightarrow a)$ wie in der Funktion
`id :: a -> a` steht abkürzend für:
 $\forall(a) a \rightarrow a$ "...für alle Typen"

- Ad hoc Polymorphie
Der Typ $(\text{Num } a \Rightarrow a \rightarrow a \rightarrow a)$ wie in der Funktion
`(+) :: Num a => a -> a -> a` steht abkürzend für:
 $\forall(a \in \text{Num}) a \rightarrow a \rightarrow a$ "...für alle Typen aus Num"

Im Haskell-Jargon ist Num eine sog.

- *Typklasse*

...eine von vielen in Haskell vordefinierten Typklassen.

Typklassen in Haskell

Informell

- Eine Typklasse ist eine Kollektion von Typen, auf denen eine in der Typklasse festgelegte Menge von Funktionen definiert ist.
- Die Typklasse Num ist die Kollektion der numerischen Typen Int, Integer, Float, etc., auf denen die Funktionen (+), (*), etc. definiert sind.

Hinweis: Vergleiche dieses Klassenkonzept z.B. mit dem Schnittstellenkonzept aus Java. Gemeinsamkeiten, Unterschiede?

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

227

Polymorphie vs. Ad hoc Polymorphie

Informell...

- (*Parametrische*) *Polymorphie* ...
~> gleicher Code trotz unterschiedlicher Typen
- *ad-hoc Polymorphie* (synonym: *Überladen* (engl. *Overloading*))...
~> unterschiedlicher Code trotz gleichen Namens (mit i.a. sinnvollerweise ähnlicher Funktionalität)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

228

Ein erweitertes Bsp. zu Typklassen (1)

Wir nehmen an, wir seien an der Größe interessiert von

- Listen und
- Bäumen

Der Begriff "Größe" sei dabei typabhängig, z.B.

- Anzahl der Elemente bei Listen
 - Anzahl der Knoten
 - Knoten
 - Blätter
 - Benennungen
 - ...
- bei Bäumen

Ein erweitertes Bsp. zu Typklassen (2)

Wir betrachten folgende Baumvarianten...

```
data Tree a = Nil |
  Node a (Tree a) (Tree a)

data Tree1 a b = Leaf1 b |
  Node1 a b (Tree1 a b) (Tree1 a b)

data Tree2 = Leaf2 String |
  Node2 String Tree2 Tree2

...und den Haskellstandardtyp für Listen.
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

230

Ein erweitertes Beispiel zu Typklassen

Naive Lösung: Schreibe für jeden Typ eine passende Funktion:

```
sizeT :: Tree a -> Int      -- Zählen der Knoten
sizeT Nil                 = 0
sizeT (Node n l r) = 1 + sizeT l + sizeT r

sizeT1 :: (Tree1 a b) -> Int -- Zählen der Benennungen
sizeT1 (Leaf1 m)         = 1
sizeT1 (Node1 m n l r) = 2 + sizeT1 l + sizeT1 r

sizeT2 :: Tree2 -> Int     -- Summe der Längen der Benennungen
sizeT2 (Leaf2 m)          = length m
sizeT2 (Node2 m l r) = length m + sizeT2 l + sizeT2 r

sizeList :: [a] -> Int    -- Zählen der Elemente
sizeList = length
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

231

Ein erweitertes Bsp. zu Typklassen (3)

"Smarter" Lösung mithilfe von Typklassen:

```
class Size a where
  size :: a -> Int

instance Size (Tree a) where
  -- Instanzbildung fuer (Tree a)
  size Nil = 0
  size (Node n l r) = 1 + size l + size r

instance Size (Tree1 a b) where
  -- Instanzbildung fuer (Tree1 a b)
  size (Leaf1 m) = 1
  size (Node1 m n l r) = 2 + size l + size r

instance Size Tree2 where
  -- Instanzbildung fuer Tree2
  size (Leaf2 m) = length m
  size (Node2 m l r) = length m + size l + size r

instance Size [a] where
  size = length
```

Ein erweitertes Bsp. zu Typklassen (4)

Das Kommando :t size liefert:

```
size :: Size a => a -> Int
size Nil => 0
size (Node "asdf" (Node "jk" Nil Nil) Nil) => 2
size (Leaf1 "asdf") => 1
size (Node1 "asdf" 3
      (Node1 "jk" 2 (Leaf1 17) (Leaf1 4))
      (Leaf1 21)) => 7
size (Leaf2 "abc") => 3
size (Node2 "asdf"
      (Node2 "jkerdt" (Leaf2 "abc") (Leaf2 "ac"))
      (Leaf "xy")) => 17
size [5,3,45,676,7] => 5
size [True,False,True] => 3
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

233

Definition von Typklassen

Allgemeines Muster einer Typklassendefinition...

```
class Name tv where
  ...signature involving the type variable tv
wobei
```

- Name ... Identifikator der Klasse
- tv ... Typvariable
- signature ... Liste von Namen zusammen mit Ihren Typen

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

234

Schlussfolgerungen zu Typklassen

Intuitiv...

...Typklassen sind Kollektionen von Typen, für die eine gewisse Menge von Funktionen ("gleicher" Funktionalität) definiert ist.

Beachte...

... "Gleiche" Funktionalität kann nicht syntaktisch erzwingen werden, sondern liegt in der Verantwortung des Programmiers!

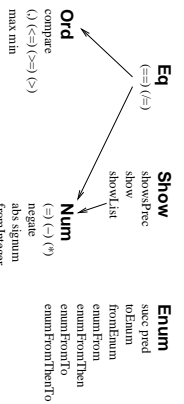
Mithin: ... Appell an die *Programmierdisziplin* unabdingbar!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

235

Bsp. einiger in Haskell vorderf. Typklassen (2)

Auswahl vorderdefinierter Typklassen, ihrer Abhängigkeiten, Operatoren und Funktionen in Standard Prelude nebst Bibliotheken:



Quelle: Fethi Rabhi, Guy Lapalme, "Algorithms - A Functional Approach", Addison-Wesley, 1999.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

237

Fortführung von Kapitel 3

Nachtzüge, Ergänzungen und Weiterführendes zu

- Ad hoc Polymorphie (Überladen)
 - Typklassen
 - Vererbung (Einfachvererbung, Mehrfachvererbung)
 - Überschreiben
 - Polymorphie vs. ad hoc Polymorphie
 - Vorteile für die Programmierung
- Listen, Muster und Funktionen
 - Listen und Muster
 - Listenkomprehension
 - Listenkonstruktoren vs. Listenoperatoren

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

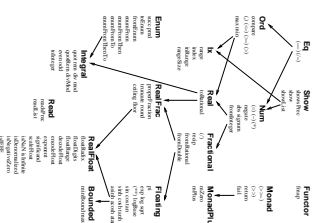
239

Typklassen: Ein größerer Ausschnitt vorderf. Typklassen

- *Gleichheit* Eq ... die Klasse der Typen mit Gleichheitstest
- *Ordnungen* Ord ... die Klasse der Typen mit Ordnungsrelationen (wie etwa <, ≤, >, ≥, etc.)
- *Aufzählung* Enum ... die Klasse der Typen, deren Werte aufgezählt werden können (Bsp.: [2,4..29])
- *Werte zu Zeichenreihen* Show ... die Klasse der Typen, deren Werte als Zeichenreihen dargestellt werden können
- *Zeichenreihen zu Werten* Read ... die Klasse der Typen, deren Werte aus Zeichenreihen herleitbar sind
- ...

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

236



Quelle: Fethi Rabhi, Guy Lapalme, "Algorithms - A Functional Approach", Addison-Wesley, 1999, Figure 2.4.

Rückblick: Typklassenbeispiel (1)

Wir hatten angenommen, an der Größe interessiert zu sein von

- Listen und
- Bäumen
- ... wobei der Begriff "Größe" typabhängig gedacht war, z.B.

- Anzahl der
 - Elemente
 - *Tupelkomponenten* (als zusätzliches Beispiel) bei (Tupel-) Listen
- Anzahl der
 - Knoten
 - Blätter
 - Benennungen
 - ...
- ... bei Bäumen

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

240

Rückblick: Typklassenbeispiel (2)

- *Wunsch*: ...eine Funktion `size`, die mit Argumenten der verschiedenen Typen aufgerufen werden kann und das gewünschte leistet.
- *Lösung*: ...ad hoc Polymorphie mittels Typklassen

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

241

Rückblick: Typklassenbeispiel (3)

Wir betrachten folgende Baumvarianten...

```
data Tree a = Nil |
  Node a (Tree a) (Tree a)

data Tree1 a b = Leaf1 b |
  Node1 a b (Tree1 a b)

data Tree2 = Leaf2 String |
  Node2 String Tree2 Tree2
```

...und den Haskell-Standardtyp für Listen.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

242

Rückblick: Typklassenbeispiel (4)

Haskells Typklassenkonzept erlaube uns folgende Lösung:

```
class Size a where
  size :: a -> Int

instance Size (Tree a) where
  size Nil
    = 0
  size (Node n l r) = 1 + size l + size r

instance Size (Tree1 a b) where
  size (Leaf1 m)
    = 1
  size (Node1 m n l r) = 2 + size l + size r

instance Size Tree2 where
  size (Leaf2 m)
    = Length m
  size (Node2 m l r) = Length m + size l + size r

instance Size [a] where
  size = Length
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

243

Rückblick: Typklassenbeispiel (5)

Mit diesen Definitionen galt für den Typ der Funktion `size`:

```
size :: Size a => a -> Int
```

...und die Funktion `size` ermöglichte wie gewünscht:

```
size Nil => 0
size (Node "asdf" (Node "jk" Nil Nil) Nil) => 2
size (Leaf1 "adf") => 1
size (Node1 "jk" 2 (Leaf1 17) (Leaf1 4))
  (Leaf1 21) => 7
size (Leaf2 "abc") => 3
size (Node2 "asdf"
  (Node2 "jkertt" (Leaf2 "abc") (Leaf2 "ac")))
  (Leaf2 "xy")
  => 17
size [5,3,45,676,7] => 5
size [True,False,True] => 3
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

244

Als zusätzliches Beispiel (1)

...sei für Tupellisten "Größe" nicht durch die Anzahl der Listenelemente, sondern durch die Anzahl der Komponenten der tupelförmigen Listenelemente gegeben.

Lösung durch entsprechende Instanzbildung:

```
instance Size [(a,b)] where
  size = (*2) . Length

instance Size [(a,b,c)] where
  size = (*3) . Length
```

Beachte: Die Instanzbildung `instance Size [(a,b)]` geht über den Standard von Haskell 98 hinaus und ist nur in entsprechenden Erweiterungen möglich.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

245

Als zusätzliches Beispiel (2)

Wie bisher gilt für den Typ der Funktion `size`:

```
size :: Size a => a -> Int
```

...und wir erhalten wie erwartet und gewünscht:

```
size [(5,"Smith"),(4,"Hall"),(7,"Douglas")] => 6
size [(5,"Smith",45),(4,"Hall",37),(7,"Douglas",42)] => 9
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

246

Anmerkungen

Sprechweisen:

- **(*2)**, **(*3)** sind im Haskell-Jargon *operator sections*.
- `"` bezeichnet in Haskell die aus der Mathematik bekannte Funktionskomposition:

Sei $f : B \rightarrow C$ und $g : A \rightarrow B$, dann ist die *Funktionskomposition* ($f \circ g$) : $A \rightarrow C$ definiert durch:

$$\forall a \in A. (f \circ g)(a) =_df f(g(a))$$

Damit bedeutet z.B.

```
((*2) . Length) [(5,"Smith"),(4,"Hall"),(7,"Douglas")]
```

dasselbe wie

```
(*2) (Length [(5,"Smith"),(4,"Hall"),(7,"Douglas")])
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

247

Wermutstropfen (1)

Die Instanzbildungen

```
instance Size [a] where
  size = Length
```

```
instance Size [(a,b)] where
  size = (*2) . Length
```

```
instance Size [(a,b,c)] where
  size = (*3) . Length
```

sind nicht gleichzeitig möglich.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

248

Wermutstropfen (2)

Problem: Überlappende Typen!

```
ERROR "est.hs:45" - Overlapping instances for class "Size"
*** This instance  : Size [(a,b)]
*** Overlaps with  : Size [a]
*** Common instance : Size [(a,b)]
```

Konsequenz:

- Für Argumente von Instanzen des Typs [(a,b)] (und ebenso des Typs [(a,b,c)]) ist die Überladung des Operators size nicht mehr auflösbar
- Wünschenswert wäre:

```
instance Size [a] without [(b,c)], [(b,c,d)] where
  size = length
```

Beachte: In dieser Weise in Haskell nicht möglich.

Zusammenfassendes

...über die Funktion size und die Typklasse Size:

- die Typklasse size stellt die Typspezifikation der Funktion size zur Verfügung
- jede Instanz der Typklasse size muss eine instanzspezifische Implementierung der Funktion size zur Verfügung stellen
- Im Ergebnis ist die Funktion size wie auch z.B. die in Haskell vordefinierten Operatoren +, *, -, etc., oder die Relatoren ==, >, >=, etc. *überladen*
- Synonym für *Überladen* ist *ad hoc Polymorphie*

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 250

Mehr zu Typklassen

Anders als die Typklasse size können Typklassen auch

- Spezifikationen mehr als einer Funktion bereitstellen
- Standardimplementierungen (engl. *default implementations*) für (alle oder einige) dieser Funktionen bereitstellen
- von anderen Typklassen *erben*

In der Folge betrachten wir dies an ausgewählten Beispielen von in Haskell vordefinierten Typklassen...

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 251

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 252

Die vordefinierte Typklasse Eq (2)

Beachte:

- die Standardimplementierungen sind für sich allein nicht ausreichend, sondern stützen sich wechselseitig aufeinander ab.

Trotz dieser Unvollständigkeit ergibt sich als Vorteil:

- Bei Instanzbildungen reicht es, entweder eine Implementierung für (==) oder für (/=) anzugeben. Für den jeweils anderen Operator gilt dann die vordefinierte Standard- (default) Implementierung.

- Auch für beide Funktionen können bei der Instanzbildung Implementierungen angegeben werden. In diesem Fall werden beide Standardimplementierungen *überschrieben*.

Übung: Vgl. dies z.B. mit Schnittstellendefinitionen und Definitionen abstrakter Klassen in Java. Welche Gemeinsamkeiten/Unterschiede gibt es?

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 254

Beispiele von Instanzbildungen der Typklasse Eq (1)

Am Beispiel des Typs der Wahrheitswerte:

```
instance Eq Bool where
  (==) True True  = True
  (==) False False = True
  (==) _ _       = False
```

Beachte: Der Ausdruck "Instanz" im Haskell-Jargon ist überladen!

- Bisherig: Typ T ist Instanz eines Typs U (z.B. Typ [Int] ist Instanz des Typs [a])
- Jetzt zusätzlich: Typ T ist Instanz einer (Typ-) Klasse C (z.B. Typ Bool ist Instanz der Typklasse Eq)

Beispiele von Instanzbildungen der Typklasse Eq (2)

Am Beispiel eines Typs für Punkte in der (x,y)-Ebene:

data Point = Point (Int,Int)

```
instance Eq Point where
  (==) (Point x) (Point y) = (fst(x)==fst(y)) &&
    (snd(x)==snd(y))
```

Bemerkung: Mit feingranularen Mustern lässt sich die Implementierung einfacher und transparenter realisieren:

```
instance Eq Point where
  (==) (Point (x,y)) (Point (u,v)) = (x==u) && (y==v))
```

Beachte: Typ- und Konstruktorkname (Point) dürfen übereinstimmen.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 255

Beispiele von Instanzbildungen der Typklasse Eq (3)

Auch selbstdefinierte Typen können zu Instanzen vordefinierter Typklassen gemacht werden. Z.B. folgender Baumtyp zur Instanz der Typklasse Eq:

```
data Tree = Nil |
  Node Int Tree Tree
instance Eq Tree where
  (==) Nil Nil      = True
  (==) (Node m t1 t2) (Node n u1 u2)
    = (m == n) &&
      (t1 == u1) &&
      (t2 == u2)
  (==) _ _         = False
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 256

Beispiele von Instanzbildungen der Typ-Klasse Eq (4)

Das Vorgenannte gilt selbstverständlich auch für selbstdefinierte polymorphe Typen wie folgende Beispiele zeigen:

```
data Tree1 a = Leaf1 a |
             Node1 a (Tree1 a) (Tree1 a)

data Tree2 a b = Leaf2 b |
               Node2 a b (Tree2 a b) (Tree2 a b)
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

257

Einschub zu Sprechweisen

```
instance (Eq a) => Eq (Tree1 a) where
  (==) (Leaf1 s) (Leaf1 t)      = (s == t)
  (==) (Node1 s t1 t1) (Node1 t u1 u2) = (s == t) &&
                                     (t1 == u1) &&
                                     (t2 == u2)
  (==) _ _                      = False
```

Sprechweisen und Vereinbarungen:

- `Tree1 a` ist Instanz der (gehört zur) Typklasse `Eq`, wenn `a` zu dieser Klasse gehört.
- Der Teil links von `=>` heißt *Kontext*.
- Rechts von `=>` dürfen ausschließlich Basistypen (z.B. `Int`), Typkonstruktoren beinhaltende Typen (z.B. `Tree a`, `[...]`) oder auf ausgezeichnete Typvariablen angewandte Tupel-typen (z.B. `(a,b,c,d)`) stehen.

Zusammenfassendes über den Relator (==) und die Typklasse Eq

Der Vergleichsoperator (`==`) ist..:

- überladen (Synonym: `ad hoc polymorph`), nicht echt polymorph
- in Haskell als Operation in der Typklasse `Eq` vorgegeben.
- damit anwendbar auf Werte aller Typen, die Instanz von `Eq` sind
- viele Typen sind bereits vordefinierte Instanz von `Eq`, z.B. alle elementaren Typen, Tupel und Listen über elementaren Typen
- auch selbstdefinierte Typen können zu Instanzen von `Eq` gemacht werden

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

261

Zur Antwort (1)

Nein!

Der Grund ist im Kern folgender:

Anders als z.B. die Länge einer Liste, die eine vom konkreten Listentyp unabhängige Eigenschaft ist und deshalb eine (echt) polymorphe Eigenschaft ist und eine entsprechende Implementierung erlaubt

```
length :: [a] -> Int           -- echt polymorph
length [] = 0
length (_:xs) = 1 + length xs
```

ist Gleichheit eine typabhängige Eigenschaft, die eine typspezifische Implementierung verlangt.

Beispiel:

- unsere typspezifischen Implementierungen des Gleichheitstests auf Bäumen

Beispiele von Instanzbildungen der Typ-Klasse Eq (5)

```
instance (Eq a) => Eq (Tree1 a) where
  (==) (Leaf1 s) (Leaf1 t)      = (s == t)
  (==) (Node1 s t1 t1) (Node1 t u1 u2) = (s == t) &&
                                     (t1 == u1) &&
                                     (t2 == u2)
  (==) _ _                      = False
```

```
instance (Eq a, Eq b) => Eq (Tree2 a b) where
  (==) (Leaf2 q) (Leaf2 s)      = (q == s)
  (==) (Node2 p q t1 t1) (Node2 r s u1 u2) = (p == r) &&
                                     (q == s) &&
                                     (t1 == u1) &&
                                     (t2 == u2)
  (==) _ _                      = False
```

Bemerkungen:

- Getrennt durch Komma wie in `(Eq a, Eq b)` können in Kontexten mehrfache `---` konjunktiv zu verstehende `---` (Typ-) Bedingungen angegeben werden.
- Damit die Anwendbarkeit des Relators (`==`) auf Werte von Knotenbenennungen gewährleistet ist, muss die Instanzen der Typvariablen `a` und `b` selbst schon als Instanz der Typklasse `Eq` vorausgesetzt sein.

Beispiele von Instanzbildungen der Typ-Klasse Eq (6)

Instanzbildungen sind flexibel...

Abweichend von der vorher definierten Gleichheitsrelation auf Bäumen vom Typ `(Tree2 a b)`, hätten wir den Gleichheitstest etwa auch so festlegen können, dass die Markierungen vom Typ `a` in inneren Knoten für den Gleichheitstest irrelevant sind:

```
instance (Eq b) => Eq (Tree2 a b) where
  (==) (Leaf2 q) (Leaf2 s)      = (q == s)
  (==) (Node2 _ q t1 t1) (Node2 _ s u1 u2) = (q == s) &&
                                     (t1 == u1) &&
                                     (t2 == u2)
  (==) _ _                      = False
```

Beachte, dass für Instanzen des Typs `a` jetzt nicht mehr Mitgliedschaft in der Typklasse `Eq` gefordert werden muss.

Funktionale Programmierkurs WS 2008/2009 (Stand: 06.11.2008)

260

Spezielle Frage

Ist es vorstellbar, jeden Typ zu einer Instanz der Typklasse `Eq` zu machen?

De facto hieße das, den Typ des Vergleichsoperators (`==`) von

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

zu

```
(==) :: a -> a -> Bool
```

zu verallgemeinern.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

262

Zur Antwort (3)

Warum ist nicht mehr möglich?

Im Sinne von Funktionen als *first class citizens* in funktionalen Sprachen wäre ein Gleichheitstest auch auf Funktionen sicher höchst wünschenswert.

Z.B.:

```
(==) fac fib           => False
(==) (\x -> x+x) (\x -> 2*x) => True
(==) (+2) doubleInc  => True
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

264

Zur Antwort (4)

In Haskell erforderte eine Umsetzung Instanzbildungen der Art:

```
instance Eq (Int -> Int) where
  (==) f g = ...
instance Eq (Int -> Int -> Int) where
  (==) f g = ...
```

Können wir die "Punkte" so ersetzen, dass wir einen Gleichheitstest für alle Funktionen der Typen `(Int -> Int)` und `(Int -> Int -> Int)` haben?

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 265

Zur Antwort (6)

Erinnerung:

"Gleichheit von Funktionen ist unentscheidbar" heißt informell, dass...

- es gibt keinen Algorithmus, der für zwei beliebig vorgelegte Funktionen stets nach endlich vielen Schritten entscheidet, ob diese Funktionen gleich sind oder nicht.

Machen Sie sich klar, dass daraus in der Tat nicht folgt, dass Gleichheit zweier Funktionen nie (in endlicher Zeit) entschieden werden kann.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 267

Schlussfolgerung (2)

Auch wenn es schön wäre, eine (echt) polymorphe Implementierung von `(==)` zu haben zur Signatur

```
(==) :: a -> a -> Bool
```

und damit analog zur Funktion zur Längenbestimmung von Listen

```
length :: [a] -> Int
```

...ist eine Implementierung in dieser Allgemeinheit für `(==)` nicht möglich!

Die Typklassen, für die eine Implementierung von `(==)` angegeben werden kann, sind in Haskell in der Typklasse `Eq` zusammengefasst.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 269

Typklassen und Vererbung (2)

- Die (wie `Eq` vordefinierte) Typklasse `Ord` erweitert die Klasse `Eq`.

- Jede Instanz der Typklasse `Ord` muss Implementierungen für alle Funktionen der Klassen `Eq` und `Ord` bereitstellen.

Beachte:

- `Ord` stellt wie `Eq` für einige Funktionen bereits Standardimplementierungen bereit.
- Bei der Instanzbildung für weitere Typen reicht es deshalb, Implementierungen der Relatoren `(==)` und `(<)` anzugeben.
- Durch Angabe instanzspezifischer Implementierungen bei der Instanzbildung können diese Standardimplementierungen aber auch nach Wunsch überschrieben werden.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 271

Zur Antwort (5)

Leider nein!

Zwar läßt sich für konkret vorgelegte Funktionen Gleichheit fallweise (algorithmisch) entscheiden, generell aber gilt folgendes aus der Theoretischen Informatik bekannte negative Ergebnis:

Theorem

Gleichheit von Funktionen ist unentscheidbar.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 266

Schlussfolgerung (1)

...anhand der Beobachtungen am Gleichheitstest `(==)`:

- ...offenbar können Funktion bestimmter Funktionalität nicht für jeden Typ angegeben werden, insbesondere läßt sich nicht für jeden Typ eine Implementierung des Gleichheitsrelators `(==)` angeben, sondern nur für eine Teilmenge aller möglichen Typen.
- ...die Teilmenge der Typen, für die das für den Gleichheitsrelator möglich ist, bzw. eine Teilmenge davon, für die das in einem konkreten Haskell-Programm tatsächlich gemacht wird, ist im Haskell-Jargon eine *Sammlung* (engl. *collection*) von Typen, eine sog. *Typklasse*.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 268

Typklassen und Vererbung (1)

Vererbung auf Typklassenebene...

```
class Eq a => Ord a where
  (<), (<=), (>), (>=) :: a -> a -> Bool
  max, min           :: a -> a -> a
  compare           :: a -> a -> Ordering
  x <= y            = (x<y) || (x==y)
  x > y             = y < x
  ...
  compare x y      = Eq
                  | x == y
                  | x <= y
                  | otherwise = GT
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 270

Typklassen und Vererbung (3)

Auch *Mehrachvererbung* auf Typklassenebene ist möglich, wie Haskell's vordefinierte Typklasse `Num` zeigt:

```
class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate      :: a -> a
  abs, signum :: a -> a
  fromInteger :: Integer -> a
  fromInt     :: Int -> a
  -- Typkonversionsfunktion!
  -- Typkonversionsfunktion!
  x - y      = x + negate y
  fromInt   = ...
```

- ...vgl. dies auch mit Vererbungskonzepten objektorientierter Sprachen!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 272

Typklassen und Vererbung (4)

Überschreiben ererbter Funktionen am Beispiel der Instanz `Point` der Typklasse `Eq`:

- *Vererbung*:
Für die Instanzdeklaration von `Point` zur Klasse `Eq`
`instance Eq Point where`
`Point (x,y) == Point (w,z) = (x==w) && (y==z)`
erbt `Point` folgende Implementierung des Ungleichheitstests (`/=`) aus der Klasse `Eq`:
`Point x /= Point y = not (Point x == Point y)`
- *Überschreiben*:
Die ererbte (Standard-) Implementierung von (`/=`) kann überschrieben werden, z.B. wie unten durch eine (geringfügig) effizientere Variante:
`instance Eq Point where`
`Point (x,y) == Point (w,z) = (x==w) && (y==z)`
`Point x /= Point y`
`= if x/=w then True else y/=z`

Typklassen und Vererbung (6)

So ist

```
data Tree a = Nil |  
  Node a (Tree a) deriving Eq  
gleichbedeutend zu:  
data Tree a = Nil |  
  Node a (Tree a) (Tree a)  
  
instance Eq a => Eq (Tree a) where  
  (==) Nil Nil = True  
  (==) (Node m t1 t2) (Node n u1 u2)  
    = (m == n) &&  
      (t1 == u1) &&  
      (t2 == u2)  
  (==) _ _ = False
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 275

Typklassen und Vererbung (5)

(*Automatisch*) *abgeleitete Instanzen* von Typklassen...

```
data Spielfarbe = Kreuz | Pik | Herz | Karo  
  deriving (Eq, Ord, Enum, Bounded, Show, Read)  
data Tree a = Nil |  
  Node a (Tree a) (Tree a)  
  deriving (Eq, Ord)
```

- Algebraische Typen können durch Angabe einer `deriving`-Klausel als Instanzen vordefinierter Klassen automatisch angelegt werden.
- Intuitiv ersetzt die Angabe der `deriving`-Klausel die Angabe einer `instance`-Klausel.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 274

Typklassen und Vererbung (7)

Analog ist

```
data Tree2 a b = Leaf2 b1 |  
  Node2 a b (Tree2 a b) deriving Eq  
gleichbedeutend zu:  
data Tree2 a b = Leaf2 b1 |  
  Node2 a b (Tree2 a b) (Tree2 a b)  
  
instance (Eq a, Eq b) => Eq (Tree2 a b) where  
  (==) (Leaf2 q) (Leaf2 s) = (q == s)  
  (==) (Node2 p q t1 t1) (Node2 r s u1 u2)  
    = (p == r) &&  
      (q == s) &&  
      (t1 == u1) &&  
      (t2 == u2)  
  (==) _ _ = False
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 276

Typklassen und Vererbung (8)

Möchten Sie hingegen Gleichheit wie folgt realisiert wissen

```
data Tree2 a b = Leaf2 b1 |  
  Node2 a b (Tree2 a b) (Tree2 a b)  
  
instance (Eq a, Eq b) => Eq (Tree2 a b) where  
  (==) (Leaf2 q) (Leaf2 s) = (q == s)  
  (==) (Node2 _ q t1 t1) (Node2 _ s u1 u2)  
    = (q == s) &&  
      (t1 == u1) &&  
      (t2 == u2)  
  (==) _ _ = False
```

geht an obiger Instanzdeklaration kein Weg vorbei:

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 277

Zum Abschluss dieses Themas

Polymorphie und *Überladen* auf Funktionen bedeuten...

- vordergründig
... ein Funktionsname kann auf Argumente unterschiedlichen Typs angewendet werden.
- präzisier
 - *Polymorphe Funktionen*...
 - * ...haben eine einzige Implementierung, die für alle (Zugelassenen/abgedeckten) Typen arbeitet (Bsp.: `length :: [a] -> Int`)
- *Überladene Funktionen*...
- * ...arbeiten für Instanzen einer Klasse von Typen mit einer für jede Instanz spezifischen Implementierung (Bsp.: `size :: Size a => a -> Int`)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 279

Typklassen und Vererbung (9)

Mehr zu Typklassen, alles zu Funktionen vordefinierter Typklassen und über Grenzen und Einschränkungen etwa automatischer Ableitbarkeit von Typklassen...

- ...in jedem guten Buch über Haskell!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 278

Vorteile des Überladens von Operatoren

- Ohne Überladen ginge es nicht ohne ausgezeichnete Namen für alle Operatoren.
- Das gäbe auch für die bekanntesten arithmetischen Operatoren, so wären insbesondere Namen der Art `+Int`, `+Float`, `*Int`, `*Float`, etc. erforderlich.
- Deren zwangweiser Gebrauch wäre nicht nur ungewohnt und unschön, sondern in der täglichen Praxis auch lästig.
- Haskell's Angebot, hier Abhilfe zu schaffen und Operatoren zu überladen, ist das Konzept der *Typklassen*.

Zum *Selbststudium*: Andere Sprachen wie z.B. ML und Opal gehen hier einen anderen Weg und bieten andere Konzepte.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 280

Kapitel 4: Listen und Listenkomprehension, Muster

Nachträge und Ergänzungen...

- Listen, Muster und Funktionen
- Listenkomprehension *at work*

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

281

Zurück zu Listen, Mustern und Funktionen darauf (1)

Einige Beispiele...

```
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs

head :: [a] -> a
head (x:_) = x

tail :: [a] -> [a]
tail (_:xs) = xs

null :: [a] -> Bool
null [] = True
null ( _: ) = False
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

282

Zurück zu Listen, Mustern und Funktionen darauf (2)

Muster, Wild Cards und Typvariablen...

```
sign :: Integer -> Integer
sign x
  | x > 0 = 1
  | x == 0 = 0
  | x < 0 = -1

takeFirst :: Integer -> [a] -> [a]
takeFirst m ys = case (m,ys) of
  (0,_) -> []
  (_,[]) -> []
  (n,x:xs) -> x : takeFirst (n - 1) xs

ifThenElse :: Bool -> a -> a -> a
ifThenElse c t e = case c of True -> t
                          False -> e
```

Oft sehr nützlich

...das sog. *as-Muster* (mit \emptyset gelesen als "as"):

Beispiel:

```
listTransform :: [a] -> [a]
listTransform l@(x:xs) = (x : 1) ++ xs
```

Zum Vergleich ohne *as-Muster*...

```
listTransform :: [a] -> [a]
listTransform (x:xs) = (x : (x : xs)) ++ xs
```

...weniger elegant und weniger gut lesbar.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

285

Somit erhalten wir als Fortschreibung...

...des Musterbegriffs: Muster können sein...

Bislang:

- *Werte* (z.B. 0, 'c', True)
...ein Argument "passt" auf das Muster, wenn es vom entsprechenden Wert ist.
- *Variablen* (z.B. n)
...jedes Argument passt (und ist rechtsseitig verwendbar).
- *Wild card* "_"
...jedes Argument passt (aber ist rechtsseitig nicht verwendbar).

Jetzt zusätzlich:

- *Konstruktormuster* (hier über Listen; z.B. [], (p:ps))
 - Eine Liste passt auf das Muster [], wenn sie leer ist.
 - Eine Liste passt auf (p:ps), wenn sie nicht leer ist und der Listenkopf auf p und der Listenrest auf ps passt.

Hinweis: Im Fall von (p:ps) reicht es, dass die Argumentliste nicht leer ist.

Ein weiteres Beispiel

Auch Funktionsdeklarationen der Form...

```
binom :: (Integer, Integer) -> Integer
binom (n,k)
  | k==0 || n==k = 1
  | otherwise   = binom (n-1,k-1) + binom (n-1,k)

...sind Beispiele musterbasierter Funktionsdefinitionen.
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

286

Zum Vergleich...

...mit Standardselektoren ohne Muster:

```
binom :: (Integer, Integer) -> Integer
binom p
  | snd(p)==0 || snd(p)==fst(p) = 1
  | otherwise = binom (fst(p)-1, snd(p)-1)
                + binom (fst(p)-1, snd(p))
```

...offenbar auch hier weniger elegant und weniger gut lesbar.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

287

Schlussfolgerung zwischendurch

Musterbasierte Funktionsdefinitionen sind (!a.)...

- elegant und
- führen zu knappen, gut lesbaren Spezifikationen.

Zu beachten aber ist: Musterbasierte Funktionsdefinitionen...

- können zu subtilen Fehlern führen und
- erschweren (oft) Programmänderungen/-weiterentwicklungen ("bis hin zur Tortur" (vgl. Pepper [4]): denke etwa an das Hinzukommen eines weiteren Parameters)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

288

Listen, Listenkonstruktoren, Listenoperatoren

Wir kennen den vordefinierten Listentyp `String`:

```
Type String = [Char]
```

... und Beispiele gültiger Werte des Typs `String`, etwa:

```
[ 'h', 'e', 'l', 'l', 'o', ' ' ] == "hello"
"hello" ++ " world" -> "hello world" (++) vordef. Konkatenationsep.)
```

Wir hatten aber auch gesehen, dass Elementtypen weit komplexer sein dürfen, bis hin zu Funktionen (Funktionen als "first class citizens"):

- Listen von Listen
[[2,4,23,2,5], [3,4], [], [56,7,6,1]] :: [[Int]]
- Listen von Paaren
[(3,14,42.0), (56,1,51.3), (1,12,2.22)] :: [Point]
- ...
- Listen von Funktionen
[fac, fib, fun91] :: [Integer -> Integer]

Ist die Zulässigkeit von [fac, fib, fun91] :: [Integer -> Integer] bemerkenswert?

Listenkomprehension (1)

...ein für funktionale Programmiersprachen charakteristisches Ausdrucksmittel.

- Listenkomprehension

...ein einfaches Beispiel:

```
[3*n | n <- [1..5]] steht kurz für [3,6,9,12], wobei [1..5] vom
```

Wert [1,2,3,4] vorausgesetzt ist.

↪ Listenkomprehension ist ein sehr elegantes und ausdruckskräftiges Sprachkonstrukt!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 291

Erinnerung aus der Mathematik...

Eine Funktion f ist ein Tripel (D, W, G) mit einer Definitionsmenge (-bereich) D , einer Wertemenge (-bereich) W und einer rechtsseitigen Relation G mit $G \subseteq D \times W$, dem sog. Funktionsgraphen von f .

Mithin...

Funktionen sind spezielle Relationen; spezielle Teilmengen eines *kartesischen Produkts*

Damit intuitiv naheliegend...

Listen von Funktionen ... "Listen von Listen von Paaren"

Schlagwort...

Funktionen als "first class citizens"

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 290

Listenkomprehension (2)

Weitere Anwendungsbeispiele:

...wobei `lst = [1,2,4,7,8,11,12,42]` vorausgesetzt ist:

- a) [square n | n <- lst] ⇒ [1,4,16,49,64,121,144,1764]
- b) [n | n <- lst, isPowOfTwo n] ⇒ [1,2,4,8]
- c) [n | n <- lst, isPowOfTwo n, n>=5] ⇒ [8]
- d) [isPrime n | n <- lst] ⇒ [False, True, False, True, False, True, False, False]

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 292

Listenkomprehension (3)

e) addCoordinates :: [Point] -> [Float]

```
addCoordinates plst = [ x+y | (x,y) <- plst, (x>0 || y>0) ]
```

```
addCoordinates [(0,0,0.5), (3,14,17.4), (-1,5,-2,3)] ⇒ [0.5,20.54]
```

f) allOdd :: [Integer] -> Bool

```
allOdd xs = ( [ x | x<-xs, isOdd x ] == xs )
```

```
allEven :: [Integer] -> Bool
```

```
allEven xs = ( [ x | x<-xs, isOdd x ] == [] )
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 293

Listenkomprehension (4)

g) grabCapVowels :: String -> String

```
grabCapVowels s = [ c | c<-s, isCapVowel c ]
```

```
isCapVowel :: Char -> Bool
```

```
isCapVowel 'A' = True
```

```
isCapVowel 'E' = True
```

```
isCapVowel 'I' = True
```

```
isCapVowel 'O' = True
```

```
isCapVowel 'U' = True
```

```
isCapVowel ' ' = False
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 294

Listenkomprehension "at work" (1)

...am Beispiel von "Quicksort".

Aufgabe: Sortiere eine Liste L ganzer Zahlen aufsteigend.

Lösung mittels Quicksort:

- **Teile:** Wähle ein Element l aus L , und partitioniere L in zwei (möglichst leere) Teillisten L_1 und L_2 so, dass alle Elemente von L_1 (L_2) kleiner (oder gleich (größer)) dem Element l sind.

- **Herrsche:** Sortiere L_1 und L_2 mit Hilfe rekursiver Aufrufe von Quicksort.

- **Zusammenführen der Teilergebnisse:** Trivial, die Gesamtliste entsteht durch Konkatenation der sortierten Teillisten.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 295

Listenkomprehension "at work" (2)

Quicksort: Eine typische Realisierung in Haskell...

```
quicksort :: [Int] -> [Int]
```

```
quicksort [] = []
```

```
quicksort (x:xs) =
```

```
quicksort [ y | y<-xs, y<=x ] ++
```

```
[x] ++ quicksort [ y | y<-xs, y>x ]
```

Beachte: Funktionsanwendung bindet stärker als Listenkonstruktion. Deshalb Klammerung des Musters `x:xs` in `quicksort (x:xs) = ...`

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 296

Listenkomprehension "at work" (3)

Zum Vergleich: Eine typische imperative Realisierung von QuickSort...

```
quickSort(L,low,high)
  if low < high
    then splitInd = partition(L,low,high)
    quickSort(L,low,splitInd-1)
    quickSort(L,splitInd+1,high) fi
partition(L,low,high)
  l = L[low]
  left = low
  for i=low+1 to high do
    if L[i] <= l then left = left+1
  swap(L[low],L[left]) fi od
return left
```

... und dem initialen Aufruf quickSort(L,1,length(L)).

Listenkonstruktoren vs. Listenoperatoren

Der Operator (:) ist Listenkonstruktor, (++) Listenoperator...

Abgrenzung: Konstruktoren führen zu eindeutigen Darstellungen, gewöhnliche Operatoren i.a. nicht.

Veranschaulicht am Beispiel von Listen:

```
42:17:4:[] == (42:(17:(4:[]))) -- eindeutig
[42,17] ++ [] ++ [4] == [42,17,4] == [42] ++ [17,4] ++ []
-- nicht eindeutig
```

Bemerkung: (42:(17:(4:[]))) deutet an, dass eine Liste ein Objekt ist, erzwungen durch die Typstruktur. Anders in imperativen/objektorientierten Sprachen: Listen sind dort nur indirekt existent, nämlich bei "geeigneter" Verbindung von Elementen durch Zeiger.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 299

Funktionale

Funktionen, unter deren Argumenten oder Resultaten Funktionen sind, heißen *Funktionen höherer Ordnung* oder kurz *Funktionale*.

Mithin...

Funktionale sind spezielle Funktionen!

...also nichts Besonderes, oder?

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 301

Aber...

The functions I grew up with, such as the sine, the cosine, the square root, and the logarithm were almost exclusively real functions of a real argument. ... I was really ill-equipped to appreciate functional programming when I encountered it: I was, for instance, totally baffled by the shocking suggestion that the value of a function could be another function.^(*)

Edsger W. Dijkstra (11.5.1930-6.8.2002)
1972 Recipient of the ACM Turing Award

^(*) Zitat aus: Introducing a course on calculi. Ankündigung einer Lehrveranstaltung an der University of Texas at Austin, 1995.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 303

Rückblick auf Kap. 1, Teil 1 der Vorlesung

Imperative vs. funktionale Programmierung – Ein Vergleich:

Gegeben ein Problem P .

Imperativ: Typischer Lösungsablauf besteht aus folgenden Schritten

1. Beschreibe eine(n) Lösung(s)weg) L für P
2. Gebe L in die Form einer Menge von Anweisungen für den Rechner und organisiere dabei die Speicherverwaltung

Funktional:

- ... das "was" statt des "wie" in den Vordergrund stellen
- ~> etwas von der Eleganz der Mathematik in die Programmierung bringen!

QuickSort: Ein eindrucksvolles Beispiel? Urteilen Sie selbst...

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 298

Kapitel 5: Funktionen höherer Ordnung

Funktionen höherer Ordnung (*kurz:* Funktionale)

- Funktionen als Argumente
 - Funktionen als Resultate
 - Spezialfall: Funktionale auf Listen
 - Anwendungen
- ...und ihre Vorteile für die Programmierung.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 300

Funktionale nichts Außergewöhnliches?

Im Grunde nicht...

Drei kanonische Beispiele aus Mathematik und Informatik:

- Mathematik: *Differential- und Integralrechnung*
 - $\frac{df(x)}{dx}$ ~> diff f a
 - ...Ableitung von f an der Stelle a
 - $\int_a^b f(x)dx$ ~> integral f a b
 - ...Integral von f zwischen a und b
- Informatik: *Semantik von Programmiersprachen*
 - Denotationelle Semantik der while-Schleife
 - $S_{a,b} \llbracket \text{while } b \text{ do } s \text{ od} \rrbracket : \Sigma \rightarrow \Sigma$
 - ...Kleinster Fixpunkt eines Funktionals auf der Menge der Zustandstransformationen $[\Sigma \rightarrow \Sigma]$ über der Menge der Zustände Σ mit $\Sigma = \text{de}\{f|\sigma \in [V^{\text{ar}} \rightarrow \text{Data}]\}$.
 - (Siehe z.B.: VU 185.183 Theoretische Informatik 2)

Feststellung

Der systematische Umgang mit Funktionen höherer Ordnung...

(Funktionen als "first-class citizens")

- ist charakteristisch für funktionale Programmierung
- hebt funktionale Programmierung von anderen Programmierparadigmen ab
- ist der Schlüssel zu extrem eleganten und ausdruckskräftigen Programmiermethoden

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 304

Ein Ausflug in die Philosophie...

Der Mensch wird erst durch Arbeit zum Menschen.

Georg W.F. Hegel (27.8.1770-14.11.1831)

Frei nach Hegel...

Funktionale Programmierung wird erst durch Funktionale zu funktionaler Programmierung!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 305

Des Pudels Kern

...bei Funktionalen:

Wiederverwendung!

...wie auch schon bei Funktionsabstraktion und Polymorphie.

Dies wollen wir in der Folge genauer herausarbeiten...

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 306

Funktionale – Motivation 1(6)

(siehe Fethi Rabhi, Guy Lapalme, Algorithms - A Functional Approach, Addison-Wesley, 1999, S. 7f.)

Betrachte folgende Beispiele...

```
-- Fakultätsfunktion
fact n | n==0 = 1
      | n>0  = n * fact(n-1)

-- Aufsummieren der n ersten natürlichen Zahlen
sumNat n | n==0 = 0
        | n>0  = n + sumNat(n-1)

-- Aufsummieren der n ersten Quadratzahlen natürlicher Zahlen
sumSqNat n | n==0 = 0
          | n>0  = n*n + sumSqNat(n-1)
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 307

Funktionale – Motivation 2(6)

Beobachtung...

- Die Definitionen von `fact`, `sumNat` und `sumSqNat` folgen demselben *Rekursionsschema*.
- Dieses zugrundeliegende gemeinsame Rekursionsschema ist gekennzeichnet durch:
 - Triff eine Festlegung für den Wert der Funktion...
 - * ...im *Basisfall*
 - * ...im verbleibenden (rekursiven) Fall als *Kombination* des Argumentwerts `n` und des Funktionswerts für `n-1`

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 308

Funktionale – Motivation 3(6)

Diese Beobachtung legt nahe...

- Obiges Rekursionsschema, gekennzeichnet durch *Basisfall* und *Funktion zur Kombination von Werten*, herauszuziehen (zu abstrahieren) und musterhaft zu realisieren.

Wir erhalten...

- Realisierung des Rekursionsschemas

```
rekPatr base comb n | n==0 = base
                   | n>0  = comb n (rekPatr base comb (n-1))
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 309

Funktionale – Motivation 4(6)

Unmittelbare Anwendung des Rekursionsschemas...

```
fact n      = rekPatr 1 (*) n
sumNat n    = rekPatr 0 (+) n
sumSqNat n  = rekPatr 0 (\x y -> x*x + y) n

...oder alternativ dazu in nichtargumentbehalteter Ausprägung:
fact      = rekPatr 1 (*)
sumNat    = rekPatr 0 (+)
sumSqNat  = rekPatr 0 (\x y -> x*x + y)
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 310

Funktionale – Motivation 5(6)

Unmittelbarer Vorteil obigen Vorgehens...

- Wiederverwendung* und dadurch...
 - *kürzterer, verlässlicherer, wartungsfreundlicherer* Code
- Erforderlich für erfolgreiches Gelingen...
- Funktionen höherer Ordnung* oder kürzer: *Funktionale*

Intuition: Funktionale sind (spezielle) Funktionen, die Funktionen als Argumente erwarten und/oder als Resultat zurückliefern.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 311

Funktionale – Motivation 6(6)

Illustriert am obigen Beispiel...

- Die Untersuchung des Typs von `rekPatr`...

```
rekPatr :: Int -> (Int -> Int -> Int) -> Int
```

 zeigt:
 - Die Funktion `rekPatr` ist ein *Funktional*!

In der Anwendungssituation des Beispiels gilt weiter...

	Wert i. Basisf. (base)	Fkt. z. Kb. v. W. (comb)
<code>fact</code>	1	(*)
<code>sumNat</code>	0	(+)
<code>sumSqNat</code>	0	\x y -> x*x + y

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 312

Funktionale, Teil 1 ...Funktionen als Argumente (1)

Anstatt zweier spezialisierter Funktionen...

```
max :: Ord a => a -> a -> a
max x y
  | x > y   = x
  | otherwise = y

min :: Ord a => a -> a -> a
min x y
  | x < y   = x
  | otherwise = y
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 313

Funktionale, Teil 1 ...Funktionen als Argumente (2)

...eine mit einem Funktions-/Prädikatsargument parametrisierte Funktion:

```
extreme :: Num a => (a -> a -> Bool) -> a -> a -> a
extreme p m n
  | p m n = m
  | otherwise = n
```

Anwendungsbeispiele:

```
extreme (>) 17 4 = 17
extreme (<) 17 4 = 4
```

Dadurch wird folgende alternative Definitionen von max und min möglich...

```
max x y = extreme (>) x y      bzw.    max = extreme (>)
min x y = extreme (<) x y      bzw.    min = extreme (<)
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 314

Funktionen als Argumente: Weitere Bsp.

(Gleichförmige) Manipulation der Marken eines benannten Baums bzw. Herausfiltern der Marken mit einer bestimmten Eigenschaft...

```
data Tree a = Nil |
             Node a (Tree a) (Tree a)

walkKndWk :: (a -> a) -> Tree a -> Tree a
walkKndWk f Nil = Nil
walkKndWk f (Node elem t1 t2) =
  (Node (f elem) (walkKndWk f t1) (walkKndWk f t2))

filterTree :: (a -> Bool) -> Tree a -> [a]
filterTree p Nil = []
filterTree p (Node elem t1 t2) =
  [ p elem = false] ++ (filterTree p t1) ++ (filterTree p t2)
  | otherwise = (filterTree p t1) ++ (filterTree p t2)
```

...mithilfe von Funktionalen, die in Manipulationsfunktion bzw. Prädikat parametrisiert sind.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 315

Funktionale, Teil 2 ...Funktionen als Resultate (1)

Auch diese Situation ist bereits aus der Mathematik vertraut...
Etwa in Gestalt der...

- Funktionskomposition (Komposition von Funktionen)

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

Bsp.:

Theorem [...bekannt aus Analysis I]

Die Komposition stetiger Funktionen ist wieder eine stetige Funktion.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 317

Funktionale, Teil 2 ...Funktionen als Resultate (3)

In komplexen Situationen einfacher zu verstehen und zu ändern als die argumentversehene Varianten...

Vergleiche folgende zwei argumentversehene Varianten der Funktion giveFourthElem :: [a] -> a ...

```
giveFourthElem ls = (head . tripleTail) ls      -- Variante 1
giveFourthElem ls = head (tripleTail ls)       -- Variante 2
```

...mit der argumentlosen Variante

```
giveFourthElem = head . tripleTail
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 319

Funktionale, Teil 2 ...Funktionen als Resultate (2)

...ermöglichen Funktionsdefinitionen auf dem (Abstraktions-) Niveau von Funktionen statt von (elementaren) Werten.

Beispiel:

```
giveFourthElem :: [a] -> a
giveFourthElem = head . tripleTail
```

```
tripleTail :: [a] -> [a]
tripleTail = tail . tail . tail
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 318

Funktionen als Resultate – Weitere Beispiele (1)

Iterierte Funktionsanwendung...

```
iterate :: Int -> (a -> a) -> (a -> a)
iterate n f
  | n > 0   = f . iterate (n-1) f
  | otherwise = id
```

```
id :: a -> a
id a = a
```

-- Anwendungsbeispiel

```
(iterate 3 square) 2
=> (square . square . square . id) 2 => 256
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 320

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 316

Zwischenresümee 1: Funktionen als Argumente...

- ...erhöhen die Ausdruckskraft erheblich und
- ...unterstützen Wiederverwendung.

Beispiel:

Vergleiche

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _ _ _ _ = []
```

mit

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _ _ _ = []
```


Funktionen als Resultate – Weitere Beispiele (2)

Anheben (engl. *lifting*) eines Wertes zu einer (konstanten) Funktion...

```
constFm :: a -> (b -> a)
constFm c = \x -> c
```

```
-- Anwendungsbeispiele
constFm 42 "Die Antwort auf alle Fragen"   => 42
constFm iterate giveFourthElem           => iterate
(constFm iterate (+) 3 (\x->x*x)) 2       => 256
```

Vertauschen von Argumenten...

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

```
-- Anwendungsbeispiel und Eigenschaft von flip
flip . flip      => id
```

Funktionen als Resultate – Partielle Auswertung (1)

Insbesondere die Spezialfälle der sog. *operator sections*:

Schlüssel: ...partielle Auswertung / partiell ausgewertete Operatoren

- (*) ...die Funktion, die ihr Argument verdoppelt.
- (2*) ...s.O.
- (42<) ...das Prädikat, das sein Argument daraufhin überprüft, größer 42 zu sein.
- (42:.) ...die Funktion, die 42 an den Anfang einer typkompatiblen Liste setzt.
- ...

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

322

Funktionen als Resultate – Partielle Auswertung (2)

Partiell ausgewertete Operatoren...

...besonders elegant und ausdruckskräftig in Kombination mit Funktionalen und Funktionskomposition.

Beispiel:

```
fancySelect :: [Int] -> [Int]
fancySelect = filter (42<) . map (*2)
```

...multipliziert jedes Element einer Liste mit 2 und entfernt anschließend alle Elemente, die kleiner oder gleich 42 sind.

```
reverse :: [a] -> [a]
reverse = foldl (flip (:)) []
```

...kehrt eine Liste um.

Bem: map, filter, foldl und flip werden in Kürze noch genauer besprochen.

Anm. zur Funktionskomposition

Beachte:

Funktionskomposition...

- ist assoziativ, d.h. $f \cdot (g \cdot h) = (f \cdot g) \cdot h = f \cdot g \cdot h$
- erfordert aufgrund der Bindungsstärke explizite Klammerung. (Bsp.: `head . tripleTail 1s` in Variante 1 von Folie 319 führt zu Typfehler.)
- sollte auf keinen Fall mit Funktionsapplikation verwechselt werden: $f \cdot g$ (*Komposition*) ist verschieden von $f\ g$ (*Applikation*)!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

324

Zwischenresümee 2: Funktionen als Resultate...

...von Funktionen (gleichberechtigt zu elementaren Werten) zuzulassen...

- ...ist der Schlüssel, Funktionen miteinander zu verknüpfen und in Programme einzubringen
- ...unterscheidet funktionale Programmierung signifikant von anderen Programmierparadigmen
- ...ist maßgeblich für die Eleganz und Ausdruckskraft und Prägnanz funktionaler Programmierung.

Damit bleibt (möglicherweise) die Frage...

- Wie erhält man funktionale Ergebnisse?

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

325

Einige Standardtechniken zur Entwicklung...

...von Funktionen mit funktionalen Ergebnissen:

- Explizit (Bsp.: *extreme*, *iterate*,...)
- Partielle Auswertung (curryfiziert vorliegender Funktionen) (Bsp.: `curriedAdd 4711 :: Int->Int`, `iterate 5 :: (a->a)->(a->a),...`)
 - Spezialfall: *operator sections* (Bsp.: `(*)`, `<2`,...)
- Funktionskomposition (Bsp.: `tail . tail . tail :: [a]->[a],...`)
- λ -Lifting (Bsp.: `constFm :: a -> (b -> a),...`)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

326

Spezialfall: Funktionale auf Listen

Typische Problemstellungen...

- Behandlung aller Elemente einer Liste in bestimmter Weise
- Herausfiltern aller Elemente einer Liste mit bestimmter Eigenschaft
- Aggregation aller Elemente einer Liste mittels eines bestimmten Operators
- ...

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

327

Standardfunktionale auf Listen

...werden in Fkt. Programmiersprachen in großer Zahl offeriert, auch in Haskell.

Drei in der Praxis besonders häufig verwendete Funktionale auf Listen sind die *Funktionalen*...

- map
- filter
- fold

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

328

Das Standardfunktional map (1)

```
-- Signatur
map :: (a -> b) -> [a] -> [b]

-- Implementierung mittels Listenkomprehension (Variante 1)
map f l s = [ f l | l <- l s ]

-- Implementierung mittels (expliziter) primitiver
-- Rekursion (Variante 2)
map f [] = []
map f (1:l:s) = f l : map f l s

-- Anwendungsbeispiel
map square [2,4,..10] = [4,16,36,64,100]
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

329

Das Standardfunktional map (2)

Einige Eigenschaften von map...

- Allgemein:

```
map (\x -> x) = \x -> x
map (f . g) = map f . map g
map f . tail = tail . map f
map f . reverse = reverse . map f
map f . concat = concat . map (map f)
map f (xs ++ ys) = map f xs ++ map f ys
```
- (Nur) für strikte f:

```
f . head = head . (map f)
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

330

Das Standardfunktional filter

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

-- Implementierung mittels Listenkomprehension
filter p l s = [ l | l <- l s, p l ]

-- Implementierung mittels (expliziter) primitiver Rekursion
filter p [] = []
filter p (1:l:s)
  | p l = 1 : filter p l s
  | otherwise = filter p l s

-- Anwendungsbeispiel
filter isPowerOfTwo [2,4,..100] = [2,4,8,16,32,64]
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

331

Das Standardfunktional fold (1)

“Falten” von rechts: foldr

```
-- Signatur (“folding from the right”)
foldr :: (a -> b -> b) -> b -> [a] -> b

-- Implementierung mittels (expliziter) primitiver Rekursion
foldr f e [] = e
foldr f e (1:l:s) = f l (foldr f e l s)

-- Anwendungsbeispiel
foldr (+) 0 [2,4,..10] = (+ 2 (+ 4 (+ 6 (+ 8 (+ 10 0))))))
                    = (2 + (4 + (6 + (8 + (10 + 0)))))) = 30
foldr (+) 0 [] = 0
```

In obiger Definition bedeuten:
f ..binäre Funktion, e...Startwert, und
(1:l:s) ...Liste der zu aggregierenden Werte.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

332

Das Standardfunktional fold (2)

Anwendungen von foldr zur Definition einiger Standardfunktionen von Haskell...

```
concat :: [[a]] -> [a]
concat l s = foldr (++) [] l s

and :: [Bool] -> Bool
and bs = foldr (&&) True bs
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

333

Das Standardfunktional fold (3)

“Falten” von links: foldl

```
-- Signatur (“folding from the left”)
foldl :: (a -> b -> a) -> a -> [b] -> a

-- Mittels (expliziter) primitiver Rekursion
foldl f e [] = e
foldl f e (1:l:s) = foldl f (f e l) l s

-- Anwendungsbeispiel
foldl (+) 0 [2,4,..10] = (+ (+ (+ (+ 0 2) 4) 6) 8) 10)
                    = (((((0 + 2) + 4) + 6) + 8) + 10) = 30
foldl (+) 0 [] = 0
```

In obiger Definition bedeuten:
f ..binäre Funktion, e...Startwert und
(1:l:s) ...Liste der zu aggregierenden Werte.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

334

Das Standardfunktional fold (4)

foldr vs. foldl – ein Vergleich:

```
-- Signatur (“folding from the right”)
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f e [] = e
foldr f e (1:l:s) = f l (foldr f e l s)

foldr f e [a1,a2,...,an]
=> a1 'f' (a2 'f' ... 'f' (an-1 'f' (an 'f' e))....)
```

```
-- Signatur (“folding from the left”)
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f e [] = e
foldl f e (1:l:s) = foldl f (f e l) l s

foldl f e [b1,b2,...,bn]
=> (...(((e 'f' b1) 'f' b2) 'f' ... 'f' bn-1) 'f' bn
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

335

Der Vollständigkeit halber

Das vorderdefinierte Funktional flip:

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

Anwendungsbeispiel: Listenreversion

```
reverse :: [a] -> [a]
reverse = foldl (flip (:)) []

reverse [1,2,3] => [3,2,1]
```

Zur Übung empfohlen: Nachvollziehen, dass reverse wie oben das Gewünschte leistet!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

336

Zwischenresümee 3

Typisch für funktionale Programmiersprachen ist...

- Elemente (Werte/Objekte) aller (Daten-) Typen sind *Objekte erster Klasse* (engl. *first-class citizens*), Das heißt: Jedes Datenobjekt kann...
 - Argument und Wert einer Funktion sein
 - in einer Deklaration benannt sein
 - Teil eines strukturierten Objekts sein

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

337

Folgendes Beispiel...

...illustriert dies sehr kompakt:

```
magicType = let
  pair x y z = z x y
  f y = pair y y
  g y = f (f y)
  h y = g (g y)
  in h (y->x)
```

Preisfragen:

- Welchen Typ hat `magicType`?
- Wie ist es Hugs möglich, diesen Typ zu bestimmen?

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

338

Zwischenresümee 4: Rechnen mit Funktionen...

Im wesentlichen folgende Quellen, Funktionen in Programme einzuführen...

- Explizite Definition im (Haskell-) Skript
- Ergebnis anderer Funktionen/Funktionsanwendungen
 - Explizit mit funktionalem Ergebnis
 - Partielle Auswertung
 - * Spezialfall: Operator sections
 - Funktionskomposition
 - λ -Lifting

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

339

Vorteile der Programmierung mit Funktionalen...

- Kürzere und i.a. einfacher zu verstehende Programme
 - ...wenn man die Semantik (insbesondere) der grundlegenden Funktionen und Funktionale (`map`, `filter`,...) verinnerlicht hat.
- Einfachere Herleitung und Beweis von Programmeigenschaften (*Stichwort*: Programmverifikation)
 - ...da man sich auf die Eigenschaften der zugrundeliegenden Funktionen abstützen kann.
- ...
- Wiederverwendung von Programmcode
 - ...und dadurch Unterstützung des *Programmierens im Großen*.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

340

Stichwort Wiederverwendung

Wesentliche Quellen für Wiederverwendung in fkt. Programmiersprachen sind...

- Polymorphie (auf Funktionen und Datentypen)
- Funktionale

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

341

Stärken funktionaler Programmierung

...resultieren aus wenigen Konzepten

- sowohl bei Funktionen
- als auch bei Datentypen

Die Ausdruckskraft ergibt sich in beiden Fällen durch die Kombination der Einzelstücke.

~> ...*das Ganze ist mehr als die Summe seiner Teile!*

Für eine detaillierte Diskussion: siehe Peter Pepper [4]

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

342

Kapitel 6: Ein- und Ausgabe

Die Behandlung von...

- Ein-/ Ausgabe in Haskell
 - ~> Einstieg in das *Monadenkonzept* von Haskell

...wird uns an die Schnittstelle von *funktionaler* und *imperativer* Programmierung führen!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

343

Hello World!

```
helloWorld :: IO ()
helloWorld = putStr "Hello World!"
```

Hello World...

- ...gewöhnlich eines der ersten Beispielprogramme in einer neuen Programmiersprache
- ...in dieser LVA erst im letzten Drittel!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

344

Ungewöhnlich?

Zum Vergleich...

Ein-/Ausgabe-Behandlung in...

- S. Thompson [3]: ...in Kapitel 18 (von 20)
- P. Pepper [4]: ...in Kapitel 21&22 (von 23)
- R. Bird [2]: ...in Kapitel 10 (von 12)
- A. J. T. Davie: "*An Introduction to Functional Programming Systems Using Haskell*", Cambridge, 1992. ...in Kapitel 7 (von 11)
- M. M. T. Chakravarty, G. C. Keller: "*Einführung in die Programmierung mit Haskell*", Pearson Studium, 2004. ...in Kapitel 7 (von 13)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

345

Zufall?

...oder ist Ein-/Ausgabe möglicherweise

- ...weniger wichtig in funktionaler Programmierung?
- ...oder in besonderer Weise problembehaftet?
- Tendenzuell letzteres...
- Ein-/Ausgabe.. führt uns an den Berührungspunkt von *funktionaler* und *imperativer* Programmierung!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

346

Rückblick...

Unsere bisherige Sicht fkt. Programmierung...

```
Eingabe ---> | Fkt. Programm | ---> Ausgabe
```

In anderen Worten...

Unsere bisherige Sicht fkt. Programmierung ist...

- *stapelverarbeitungsfokussiert*
- *nicht dialog- und interaktionsorientiert*
- ...wie es heutigen Anforderungen und heutiger Programmierrealität entspricht.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

347

Erinnerung

Im Vergleich zu anderen Paradigmen...

- Das funktionale Paradigma betont das "was" (Ergebnisse) zugunsten des "wie" (Art der Berechnung der Ergebnisse)

Von zentraler Bedeutung dafür...

- Der Wert eines Ausdrucks hängt nur von den Werten seiner Teilausdrücke ab
Stichwort(θ): ...Kompositionalität (ref. Transparenz)
~> erleichtert Programmentwicklung und Korrektheitsüberlegungen
- Auswertungsabhängigkeiten, nicht aber Auswertungsreihenfolge dezidiert festgelegt
Stichwort(e): ..Flexibilität (Church-Rosser-Eigenschaft)
~> erleichtert Implementierung einschl. Parallelisierung
- ...

Angenommen...

...wir hätten Konstrukte der Art (*Achtung*: Kein Haskell!)

```
PRINT :: String -> a -> a
PRINT message value =
  << << gib "message" am Bildschirm aus und liefere >>
  value
```

```
READFL :: Float
READFL = << lies Gleitkommazahl und liefere diese als Ergebnis >>
```

Mithin:

...Hinzunahme von Ein-/Ausgabe mittels seiteneffektbehafter Funktionen!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

349

Knackpunkt 1: Kompositionalität (1)

Vergleiche...

```
val :: Float
val = 3.14
```

```
valDiff :: Float
valDiff = val - val
```

mit...

```
readDiff :: Float
readDiff = READFL - READFL
```

und der Anwendung in...

```
constFunOrNot :: Float
constFunOrNot = valDiff + readDiff
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

350

Knackpunkt 1: Kompositionalität (2)

Beachte: ...der Wert von Ausdrücken hinge nicht länger nur von seinen Teilausdrücken ab (sondern auch von der Position im Programm)

~> ...Verlust von *Kompositionalität*

(...und der damit einhergehenden positiven Eigenschaften).

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

351

Knackpunkt 2: Flexibilität (1)

...oder der Verlust der Unabhängigkeit von der Auswertungsreihenfolge

Vom "Punkt" ...

```
punkt r =
```

```
let
```

```
  myP1 = 3.14
```

```
  x = r * myP1
```

```
  y = r + 17.4
```

```
  z = r * r
```

```
in (x,y,z)
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

352

Knackpunkt 2: Flexibilität (2)

...zum "Knackpunkt" (Achtung: Kein Haskell!):

```
knackpunkt r =
  let
    myPi = PRINT "Constant Value" 3.14
    u     = PRINT "Erstgelesener Wert" dummy
    c     = READFL
    x     = r * c
    v     = r * c
    d     = READFL
    y     = r + d
    z     = r * r
  in (x,y,z)
```

~> ... Verlust der *Auswertungsreihenfolgenunabhängigkeit*

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 353

Haskells Ansatz zur Behandlung von Ein- und Ausgabe

Zentral...

- Elementare Ein-/Ausgabeoperationen (Kommandos) auf speziellen Typen (IO-Typen) sowie
 - (Kompositions-) Operatoren, um Anweisungssequenzen (Kommandosequenzen) auszudrücken
- Damit...
- Trennung von
 - funktionalem Kern und
 - imperativähnlicher Ein-/Ausgabe
- ...somit gelangen wir an die Schnittstelle von funktionaler und imperativer Welt!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 355

Kompositionsooperatoren

```
(>>) :: IO a -> IO b -> IO b
(>>=) :: IO a -> (a -> IO b) -> IO b
```

Intuitiv:

- (>>) (oft *gelesen als 'sequence'*): Wenn p und q Kommandos sind, dann ist p >> q das Kommando, das zunächst p ausführt, den Rückgabewert (x vom Typ a) ignoriert, und anschließend q ausführt.

- (>>=) (oft *gelesen als 'then'* oder 'bind'): Wenn p und q Kommandos sind, dann ist p >>= q das Kommando, das zunächst p ausführt, dabei den Rückgabewert x vom Typ a liefert, und daran anschließend q x ausführt und dabei den Rückgabewert y vom Typ b liefert.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 357

Weitere Ein-/Ausgabeoperationen

```
-- Schreiben und Lesen von Werten unterschiedlicher Typen
print :: Show a => a -> IO ()
print = putStrLn . show

read :: Read a => String -> a

-- Rueckgabewerterzeugung ohne Ein-/Ausgabe (Aktion)
return :: a -> IO a
```

Erinnerung:

```
show :: Show a => a -> String
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 359

Ergo...

Konzentration auf die *Essenz* der Programmierung wie im funktionalen Paradigma ("was" statt "wie") ist wichtig und richtig, aber...

- Kommunikation mit dem Benutzer (bzw. der Außenwelt) muss die zeitliche Abfolge von Aktivitäten ausdrücken gestalten.

In den Worten von P. Pepper [4]:

- "...*der Benutzer lebt in der Zeit und kann nicht anders als zeitabhängig sein Programm beobachten*".

Konsequenz ...man (bzw. ein jedes Paradigma) darf von der Arbeitsweise des Rechners, nicht aber von der des Benutzers abstrahieren!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 354

(Ausgewählte) elementare Ein-/Ausgabeoperationen

```
-- Eingabe
getChar :: IO Char
getLine :: IO String
```

```
-- Ausgabe
putChar :: Char -> IO ()
putStrLn :: String -> IO ()
putStr  :: String -> IO ()
```

Bemerkung:

- () : ...spezieller einelementiger Haskell-Typ, dessen einziges Element (ebenfalls) mit () bezeichnet wird.
- IO a : ...spezieller Haskell-Typ "*I/O Aktion (Kommando) vom Typ a*". IO: ...Typkonstruktor (ähnlich wie [a] für Listen oder -> für Funktionstypen)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 356

Einfache Anwendungsbeispiele

```
-- Schreiben mit Zeilenvorschub (Standardoperation in Haskell)
putStrLn :: String -> IO ()
putStrLn = putStrLn . (++ "\n")
```

```
-- Lesen einer Zeile und Ausgeben der gelesenen Zeile
echo :: IO ()
echo = getLine >>= putStrLn
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 358

Die do-Notation: Bequemere (Kommando-) Sequenzenbildung

Komfortabler als (>>) und (>>=) ist Haskells *do*-Notation...

```
putStrLn :: String -> IO ()
putStrLn str = do putStrLn str
                putStrLn "\n"

putTwice :: String -> IO ()
putTwice str = do putStrLn str
                putStrLn str

putTimes :: Int -> String -> IO ()
putTimes n str = if n <= 1
  then putStrLn str
  else do putStrLn str
         putTimes (n-1) str
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 360

Weitere Beispiele zur do-Notation

```
putTwice = putStrLnTimes 2

read2Lines :: IO ()
read2Lines = do getLine
               getLine
               putStrLn "Two lines read."

echo2Times :: IO ()
echo2Times = do line <- getLine
               putStrLn line
               putStrLn line

getInt :: IO Int
getInt = do line <- getLine
         return (read line :: Int)
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

361

“Single vs. updatable Assignment” (1)

Durch das Konstrukt

```
var <- ...
```

wird stets eine *frische* Variable eingeführt.

Sprechweise:

...Unterstützung des Konzepts des

- "single assignment", nicht das des
- "updatable assignment" (destruktive Zuweisung wie aus imperativen Programmiersprachen bekannt)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

362

“Single vs. updatable Assignment” (2)

...zur Illustration des Unterschieds betrachte:

```
goUntilEmpty :: IO ()
goUntilEmpty = do line <- getLine
               while (return (line /= []))
                 (do putStrLn line
                  line <- getLine
                  return ())

wobei while :: IO Bool -> IO () -> IO ()
```

Abhilfe: ...Rekursion statt Iteration!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

363

“Single vs. updatable Assignment” (3)

...z.B. auf folgende in S. Thompson [3] auf S. 393 vorgeschlagene Weise:

```
goUntilEmpty :: IO ()
goUntilEmpty =
  do line <- getLine
   if (line == [])
   then return ()
   else (do putStrLn line
         goUntilEmpty)
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

364

Stichwort: Iteration

```
while :: IO Bool -> IO () -> IO ()

while test action
= do res <- test
  if res then do action
              while test action
            else return ()

-- "null I/O-action"
Erinnerung:
-- Rueckgabewert erzeugung ohne Ein-/Ausgabe (aktion)
return :: a -> IO a
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

365

Ein-/Ausgabe von und auf Dateien

...auch hierfür vordefinierte Standardoperatoren.

```
readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()

where
  type FilePath = String
  -- Implementationsabhängig
  -- Anwendungsbeispiel: Bestimmung der Laenge einer Datei
  size :: IO Int
  size = do putStrLn "Dateiname = "
         name <- getLine
         text <- readFile name
         return (length(text))
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

366

Zusammenhang do-Konstrukt und (>>), (>>=)-Operatoren

...illustriert anhand eines Beispiels:

Mittels do...

```
incrementInt :: IO ()
incrementInt = do line <- getLine
               putStrLn (show (1 + read line :: Int))

Äquivalent dazu...
```

```
incrementInt = getLine >> \line ->
               putStrLn (show (1 + read line :: Int))
```

Intuitiv...

"do = (>>=) plus anonyme Lambda-Abstraktion"

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

367

Konvention in Haskell

- Hauptdefinition (übersetzter) Haskell-Programme ist (per Konvention) eine Definition `main` vom Typ `IO a`.

Beispiel:

```
main :: IO ()
main = do c <- getChar
         putchar c
```

...`main` ist Startpunkt eines (übersetzten) Haskell-Programms. (Intuitiv gilt somit: "Programm = Ein-/Ausgabekommando")

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

368

Fazit über Ein- und Ausgabe

Es gilt...

- Ein-/Ausgabe grundsätzlich unterschiedlich in funktionaler und imperativer Programmierung
- Am augenfälligsten:
- *Imperativ*: Ein-/Ausgabe prinzipiell an jeder Programmstelle möglich
 - *Funktional*: Ein-/Ausgabe an bestimmten Programmstellen konzentriert
- Häufige Beobachtung...
- ...die vermeintliche Einschränkung erweist sich oft als Stärke bei der Programmierung im Großen!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

369

Ausblick

Das allgemeinere Konzept der

- Monaden in Haskell

und ihr Zusammenhang mit der Realisierung von

- Ein-/Ausgabe in Haskell

...später (Kapitel 8)!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

370

Kapitel 7: Fehlerbehandlung

... in Haskell:

- Bislang von uns nur rudimentär behandelt.
- Typische Formulierung aus den Aufgabenstellungen:
- ...so hat die Funktion als Ergebnis den aktualisierten Strimmzettel; ansonsten ist das Ergebnis die Zeichenreihe Ungültige Eingabe.
- In der Folge Wege zu einem systematischeren Umgang mit unerwarteten Programmsituationen und Fehlern

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

371

Typische Fehlersituationen

- Division durch 0
 - Zugriff auf das erste Element einer leeren Liste
 - ...
- In der Folge...
- 3 Varianten zum Umgang mit solchen Situationen

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

372

Variante 1: Panikmodus (1)

- *Berechnung anhalten und Fehlerursache melden.*

Hilfsmittel: Die Funktion `error`...

```
error :: String -> a
```

Aufruf von...

```
error "Unbehebbarer Fehler aufgetreten..."
```

liefert Ausgabe...

```
Program error: Unbehebbarer Fehler aufgetreten...
```

und Programmausführung stoppt.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

373

Variante 1 (2)

Vor- und Nachteile von Variante 1:

- Schnell und einfach

- *Aber*: Die Berechnung stoppt unwiderruflich. Jegliche (auch) sinnvolle Information über den Programmablauf ist verloren.

Ziel: Kein *Panikmodus*. Programmablauf nicht gänzlich abbrechen.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

374

Variante 2: Dummy-Werte (1)

- *Verwendung von dummy-Werten im Fehlerfall.*

Stattd...

```
tail1 :: [a] -> [a]
```

```
tail1 (_,xs) = xs
```

```
tail1 [] = error "'Preludelist.tail: empty list''
```

benutze zum Beispiel...

```
tl :: [a] -> [a]
```

```
tl (_,xs) = xs
```

```
tl [] = []
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

375

Variante 2 (2)

Vor- und Nachteile von Variante 2:

- Programmablauf wird nicht abgebrochen

- *Aber*: Ein geeigneter Default-Wert ist nicht immer offensichtlich.

Betrachte z.B.:

```
hd :: [a] -> a
```

```
hd (x:-) = x
```

```
hd [] = ????????????
```

Möglicher Ausweg:

- ...Jeweils gewünschten Wert als Parameter mitgeben.

Im obigen Beispiel etwa:

```
hdy :: a -> [a] -> a
```

```
hdy y (x:-) = x
```

```
hdy y [] = y
```

Variante 2 (3)

Generelles Muster:

Ersetze übliche Implementierung einer (einstelligen) Funktion

f...

```
f x = ...
```

durch...

```
fErr y x      = y
  | cond      = y
  | otherwise = f x
```

mit cond Charakterisierung der Fehlerituation.

Vor- und Nachteile:

- Generell, stets anwendbar
- Auftreten des Fehlerfalls nicht beobachtbar: y mag auch als gewöhnliches Ergebnis auftreten

Variante 3: Spezielle Fehlerwerte und -typen (1)

- Fehlerwerte und -typen statt *schlichter dummy*-Werte

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

...i.w. der Typ a mit dem Zusatzwert Nothing.

Damit...

```
fErr x
  | cond      = Nothing
  | otherwise = Just (f x)
```

...und anhand eines konkreten Beispiels:

```
errDiv :: Int -> Int -> Maybe Int
errDiv n m
  | (m == 0) = Nothing
  | otherwise = Just (n `div` m)
```

Variante 3 (2)

Vor- und Nachteile von Variante 3:

- Geänderte Funktionalität: statt a, jetzt Maybe a
- *Aber*:
 - Fehlerursachen können durch einen Funktionsaufruf "hindurchgereicht" werden (der Effekt der Funktion mapMaybe...)
 - Fehler können "gefangen" werden (die Rolle von der Funktion maybe...)

Variante 3 (3)

Die Funktionen mapMaybe und maybe:

```
mapMaybe :: (a -> b) -> Maybe a -> Maybe b
```

```
mapMaybe g Nothing = Nothing
mapMaybe g (Just x) = Just (g x)
```

```
maybe :: b -> (a -> b) -> Maybe a -> b
```

```
maybe n f Nothing = n
maybe n f (Just x) = f x
```

Variante 3 (4)

Anwendungsbeispiel(e):

Der Fehler wird "(auf-) gefangen":

```
maybe 42 (+1) (mapMaybe (*3) errDiv 9 0)
=> maybe 42 (+1) (mapMaybe (*3) Nothing)
=> maybe 42 (+1) Nothing
=> 42
```

Kein Fehlerfall, "alles geht gut":

```
maybe 42 (+1) (mapMaybe (*3) errDiv 9 1)
=> maybe 42 (+1) (mapMaybe (*3) (Just 9))
=> maybe 42 (+1) (Just 27)
=> 1 + 27
=> 28
```

Variante 3 (5)

Wesentlicher Vorteil der letzten Variante:

- Systementwicklung ohne explizite Fehlerbehandlung möglich.
- Fehlerbehandlung kann am Ende mithilfe der Funktionen mapMaybe und maybe ergänzt werden.
- ...für weitere Details siehe S. Thompson [3], Kapitel 14.3.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

381

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

382

Kapitel 8: Monaden

Hintergrund und Grundlagen...

- Programmierung mit Monaden
 - Zusammenhang Monaden und Ein-/Ausgabe in Haskell

Ein-/Ausgabe in Haskell und Monaden

Ein-/Ausgabe in Haskell...

- realisiert als Spezialfall eines allgemeineren Konzepts, des Konzepts der *Monade*.

In der Folge deshalb: Haskell's Monadenkonzept in größerem Detail

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

383

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

384

Monaden und Monadischer Programmierstil (1)

Monaden...

- erlauben die Reihenfolge, in der Operationen ausgeführt werden, explizit festzulegen.

Beispiel:

```
a-b -- Keine Festlegung der Auswertungsreihenfolge;
-- kritisch, falls z.B. Ein-/Ausgabe involviert ist.
do a <- getInt -- Reihenfolge explizit festgelegt
  b <- getInt
  return (a-b)
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

385

Monaden (1)

...sind *Konstruktor*klassen, Familien von Typen m a über einem polymorphen Typkonstruktor m mit den Funktionen ($>>=$), `return`, ($>>$) und `fail`.

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

m >> k = m >>= \_ -> k      -- vorderdefiniert
fail s = error s          -- vorderdefiniert
```

...wobei die Implementierungen der Funktionen gewissen Anforderungen genügen müssen.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

386

Monaden (2)

Zentral:

- Die Funktionen ($>>=$) und `return` (und das abgeleitete (und in der Anwendung oft bequemere) `do`-Konstrukt)

...für die die Konstruktor**klasse** `Monad` m keine Standardimplementierung vorsieht.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

387

Monaden (3)

Anforderungen (Gesetze) an die Monadenoperationen:

```
return a >>= f      = f a
c >>= return       = c
```

```
c >>= (\x -> (f x) >>= g) = (c >>= f) >>= g
```

Intuitiv:

- `return` gibt den Wert zurück, ohne einen weiteren Effekt.
- durch $>>=$ gegebene Sequenzierungen sind unabhängig von der Klammerung (assoziativ)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

388

Konstruktor**klassen** vs. Typ**klassen**

Im Grundsatz ähnliche Konzepte, wobei...

- Konstruktor**klassen**
 - ...haben Typkonstruktoren als Elemente
- Typ**klassen** (Eq, Ord, Num, ...)
 - ...haben Typen als Elemente
- Typkonstruktoren sind...
 - ...Funktionen, die aus gegebenen Typen neue Typen erzeugen (Bsp.: Tupelkonstruktor `()`, Listenkonstruktor `[]`, Funktionskonstruktor `->`, aber auch: Ein-/Ausgabe `IO...`)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

389

Der abgeleitete Operator ($>>=$)...

...ist folgendermaßen definiert:

```
(>>=) :: Monad m => (a -> m b) ->
      (b -> m c) ->
      (a -> m c)
```

```
f >>= g = \x -> (f x) >>= g
```

Hinweis:

...`return` ist vom Typ `a -> m a`!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

390

Damit...

```
(1) return >>= f = f
(2) f >>= return = f
(3) (f >>= g) >>= h = f >>= (g >>= h)
```

Intuitiv...

- (1)&(2): `return` ist Einselement von ($>>=$)
- (3): ($>>=$) ist assoziativ

Beachte:

Obige Eigenschaften gelten nicht a priori, sondern sind durch die Implementierung sicherzustellen!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

391

Beispiele von Monaden (1)

Die Identitätsmonade (mehr dazu auf Folie 402, Kapitel 8):

...einfachste aller Monaden.

```
(>>=) :: m a -> (a -> m b) -> m b
m >>= f = f m
```

```
return :: a -> m a
return = id
```

Erinnerung:

- ($>>$) und `fail` implizit durch die Standarddefinition festgelegt.

Bemerkung:

- In diesem Szenario...
($>>=$) wird Vorwärtskomposition von Funktionen, ($<.>$).
Beachte: ($<.>$) ist assoziativ mit Einselement `id`.

Beispiele von Monaden (2)

Die Listemonade:

```
instance Monad [] where
  xs >= f = concat (map f xs)
  return x = [x]
  fail s = []
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

393

Beispiele von Monaden (3)

Die "Maybe"-Monade:

```
instance Monad Maybe where
  (Just x) >>= k = k x
  Nothing >>= k = Nothing
  return      = Just
  fail s      = Nothing
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

394

Beispiele von Monaden (4)

Die Ein-/Ausgabe-Monade:

```
instance Monad IO where
  (>>=) :: IO a -> (a -> IO b) -> IO b
  return :: a -> IO a
  -- Rueckgabewerterzeugung
  -- ohne Ein-/Ausgabe(aktion)
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

395

Standardfunktionen über Monaden

Kombination von Monaden...

- ...zum Aufbau komplexerer Effekte

```
mapF :: Monad m => (a -> b) -> m a -> m b
mapF f m = do x <- m
          return (f x)
```

```
joinM :: Monad m => m (m a) -> m a
joinM m = do x <- m
          x
```

Bemerkung:

- Aus (1), (2) und (3) folgt:
(4) $\text{mapF } (f \cdot g) = \text{mapF } f \cdot \text{mapF } g$

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

396

Ein-/Ausgabe und Monaden

Einbringung:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
return :: a -> IO a
```

wobei...

- (>>=): Wenn p und q Kommandos sind, dann ist p >>= q das Kommando, das zunächst p ausführt, dabei den Rückgabewert x vom Typ a liefert, und daran anschließend q x ausführt und dabei den Rückgabewert y vom Typ b liefert.

- return: Rückgabewerterzeugung ohne Ein-/Ausgabe(aktion)
- Somit...

- Ein-/Ausgabe in Haskell Monad über dem Typkonstruktor IO

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

397

Programmieren mit Monaden (1)

Gegeben:

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

Aufgabe:

- Schreibe eine Funktion, die die Summe der Werte der Marken in einem Baum vom Typ Tree Int berechnet.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

398

Programmieren mit Monaden (2)

Lösung 1: Monadenlos

```
sTree :: Tree Int -> Int
sTree Nil = 0
sTree (Node n t1 t2) = n + sTree t1 + sTree t2
```

Beachte:

- Die Reihenfolge der Berechnung ist weitgehend nicht festgelegt (Freiheitsgrade!)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

399

Programmieren mit Monaden (3)

Lösung 1: Monadenbehaftet

```
sumTree :: Tree Int -> Id Int
sumTree Nil = return 0
sumTree (Node n t1 t2) = do num <- return n
                          s1 <- sumTree t1
                          s2 <- sumTree t2
                          return (num + s1 + s2)
```

...wobei Id die Identitätsmonade bezeichnet.

Beachte:

- Die Reihenfolge der Berechnung explizit festgelegt (keine Freiheitsgrade!)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

400

Programmieren mit Monaden (4)

Die Identitätsmonade:

```
data Id a = Id a
```

```
instance Monad Id where
  (>=>) (Id x) f = f x
  return      = Id
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

401

Programmieren mit Monaden (5)

Vergleich der monadenlosen und monadenbehandelten Lösung:

Es gilt:

- Anders als `stree` hat `sumTree` einen "imperativen Anstrich" in etwa vergleichbar mit:

```
num    := n;
sum1  := sumTree t1;
sum2  := sumTree t2;
return (num + sum1 + sum2);
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

402

Programmieren mit Monaden: Ein Re-sümee (1)

Die Programmierung mit Monaden erlaubt..:

- Berechnungsabläufe zu strukturieren.
- Folgende Eigenschaften prädestinieren Monaden dafür in besonderer Weise:
 - *Wohldefiniert*: ...Strategie sequentielle Programmteile systematisch zu spezifizieren.
 - *Angemessen*: ...höhere Abstraktion durch Entkopplung der zugrundeliegenden Monade von der Struktur der Berechnung.
 - *Fundiert*: ...Eigenschaften wie etwa (4) werden impliziert von den Monadforderungen (1), (2) und (3).

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

403

Programmieren mit Monaden: Ein Re-sümee (2)

Monaden sind...

- ein in der Kategorientheorie geprägter Begriff
~> ...zur formalen Beschreibung der Semantik von Programmiersprachen (Eugenio Moggi, 1989)
- (ohne obigen Hintergrund) populär in der Welt funktionaler Programmierung, insbesondere weil (Philipp Wadler, 1992)
 - erlauben gewisse Aspekte imperativer Programmierung in die funktionale Programmierung zu übertragen
 - eignen sich insbesondere zur Integration von Ein-/Ausgabe, aber auch für weitergehende Anwendungsszenarien
 - geeignete Schnittstelle zwischen funktionaler und effektiverer, z.B. imperativer und objektorientierter Programmierung.

Kapitel 9: Ausdrücke, Auswertung von Ausdrücken, Auswertungsstrategien

- Ausdrücke, Auswertung von Ausdrücken, Auswertungsstrategien
Schlagerwörter: applicative und normal order evaluation, eager und lazy evaluation, ...

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

405

Auswerten von Ausdrücken und Funktionsaufrufen

```
e :: Float
e = 2.71828

res = 2 * e * e => ...

simple           :: Int->Int->Int->Int
simple x y z     = (x + z) * (y + z)

simple 2 3 4 => ...

fac           :: Int -> Int
fac n        = if n == 0 then 1 else n * fact (n - 1)

fac 2 => ...
```

Auswerten von Ausdrücken

Einfache Ausdrücke

```
3 * (9 + 5) => 3 * 14 => 42
```

oder

```
3 * (9 + 5) => 3 * 9 + 3 * 5 => 27 + 3 * 5 => 27 + 15 => 42
```

oder ...

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

407

Auswerten von Funktionsaufrufen (1)

Fundamental...

- *Expandieren*
- *Simplifizieren*

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

408

Auswerten von Funktionsaufrufen (2)

Beispiele:

simple x y z = (x + z) * (y + z)

simple 2 3 4 ⇒ (2 + 4) * (3 + 4) (Expandieren)

⇒ 6 * (3 + 4) (Simplifizieren)

⇒ 6 * 7 (Simplifizieren)

⇒ 42 (Simplifizieren)

oder

simple 2 3 4 ⇒ (2 + 4) * (3 + 4) (Expandieren)

⇒ (2 + 4) * 7 (Simplifizieren)

⇒ 6 * 7 (Simplifizieren)

⇒ 42 (Simplifizieren)

oder ...

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 409

Funktionsaufrufe

fac n = if n == 0 then 1 else (n * fac (n - 1))

fac 2 ⇒ if 2 == 0 then 1 else (2 * fac (2 - 1))

⇒ 2 * fac (2 - 1)

Weiter mit a)

⇒ 2 * fac 1

⇒ 2 * (if 1 == 0 then 1 else (1 * fac (1-1)))

⇒ analog fortführen...

...oder mit b)

⇒ 2 * (if (2-1) == 0 then 1 else ((2-1) * fac ((2-1)-1)))

⇒ analog fortführen...

Schließlich...

⇒ 2

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 410

Auswertung gemäß Variante a)

fac n = if n == 0 then 1 else (n * fac (n - 1))

fac 2 ⇒ if 2 == 0 then 1 else (2 * fac (2 - 1))

⇒ 2 * fac (2 - 1)

⇒ 2 * fac 1

⇒ 2 * (if 1 == 0 then 1 else (1 * fac (1 - 1)))

⇒ 2 * (1 * fac (1 - 1))

⇒ 2 * (1 * fac 0)

⇒ 2 * (1 * (if 0 == 0 then 1 else (0 * fac (0 - 1))))

⇒ 2 * (1 * 1)

⇒ 2 * 1

⇒ 2

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 411

Auswertung gemäß Variante b)

fac n = if n == 0 then 1 else (n * fac (n - 1))

fac 2 ⇒ if 2 == 0 then 1 else (2 * fac (2 - 1))

⇒ 2 * fac (2 - 1)

⇒ 2 * (if (2-1) == 0 then 1 else ((2-1) * fac ((2-1)-1)))

⇒ 2 * ((2-1) * fac ((2-1)-1))

⇒ 2 * (1 * fac ((2-1)-1))

⇒ 2 * (1 * (if ((2-1)-1) == 0 then 1

⇒ 2 * (1 * (if ((2-1)-1) * fac (((2-1)-1)-1)))

⇒ 2 * (1 * 1)

⇒ 2 * 1

⇒ 2

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 412

Freiheitsgrade...

Betrachte...

```
fac(fac(square(2+2))) * fact(fac(square(3)))
```

Zentral...

- **Wo** im Ausdruck mit der Auswertung fortfahren?

- **Wie** mit (Funktions-) Argumenten umgehen?

Gretchenfrage...

- **Welcher** Einfluss auf das Ergebnis?

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 413

Glücklicherweise...

Theoremn

Jede terminierende Auswertungsreihenfolge endet mit demselben Ergebnis.

...Alonzo Church/J. Barclay Rosser (1936)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 414

Auswertungsstrategien...

In der Praxis...

Um den Ausdruck $f(exp)$ auszuwerten...

- ...berechne zunächst den Wert von exp (und setze diesen Wert dann im Rumpf von f ein)

~> applicative order evaluation, eager evaluation, call-by-value evaluation, leftmost-innermost evaluation

- ...setze exp unmittelbar im Rumpf von f ein und werte den so entstehenden Ausdruck aus

~> normal order evaluation, call-by-name evaluation, leftmost-outermost evaluation

~> "Intelligente" Realisierung: lazy evaluation, call-by-need evaluation

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 415

Ein Beispiel...

Einige einfache Funktionen...

```
-- Die Funktion square zur Quadrierung einer ganzen Zahl
```

```
square :: Int -> Int
```

```
square n = n*n
```

```
-- Die Funktion first zur Projektion auf die erste PaarKomponente
```

```
first :: (Int,Int) -> Int
```

```
first (m,n) = m
```

```
-- Die Funktion infiniteInc zum "ewigen" Inkrement
```

```
infiniteInc :: Int
```

```
infiniteInc = 1 + infiniteInc
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 416

Auswertung gemäß...

... applicative order (leftmost-innermost):

```
square(square(square(2)))
⇒ square(square(2 * 2))
⇒ square(square(4))
⇒ square(4 * 4)
⇒ square(16)
⇒ 16 * 16
⇒ 256
```

... 6 Schritte.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

417

Auswertung gemäß...

... normal order (leftmost-outermost):

```
square(square(square(2)))
⇒ square(square(2)) * square(square(2))
⇒ (square(2) * square(2)) * square(square(2))
⇒ ((2 * 2) * square(2)) * square(square(2))
⇒ (4 * square(2)) * square(square(2))
⇒ (4 * (2 * 2)) * square(square(2))
⇒ (4 * 4) * square(square(2))
⇒ 16 * square(square(2))
⇒ ...
⇒ 16 * 16
⇒ 256
```

... 1+6+6+1=14 Schritte.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

418

Applicative order effizienter?

Nicht immer...

```
first (42, square(square(2)))
```

...in applicative order

```
first (42, square(square(2)))
⇒ ...
⇒ first (42, 256)
⇒ 42
```

...1+6+1=8 Schritte.

...in normal order

```
first (42, square(square(2)))
⇒ 42
```

... 1 Schritt.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

419

Von Normal Order zu Lazy Evaluation...

- *Problem:* Mehrfachauswertung von Ausdrücken bei *normal order Evaluation*

- *Ziel:* Vermeidung von Mehrfachauswertungen zur Effizienzsteigerung

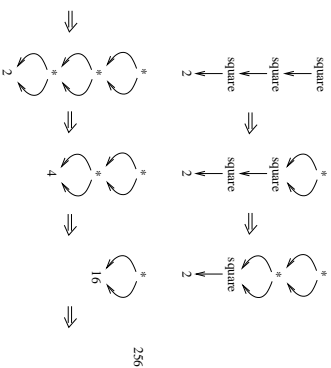
- *Lösung:* *Lazy Evaluation!*

...Ausdrucksdarstellung und -auswertung basierend auf Graphen und Graphtransformationen.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

420

Lazy evaluation (call-by-need)...



... 6 Schritte, aber Graphtransformationen!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

421

Lazy evaluation (call-by-need)...

- ...beruht (implementierungstechnisch) auf Graphtransformationen

- ...garantiert, dass Argumente höchstens einmal (möglicherweise also gar nicht) ausgewertet werden

Insgesamt...

~> ...effiziente Realisierung der normal order Strategie!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

422

Zentrale Ergebnisse...

Theorem

- ...alle terminierenden Auswertungsreihenfolgen enden mit demselben Ergebnis
~> Konfluenz- oder Diamanteigenschaft
- ...wenn es eine terminierende Auswertungsreihenfolge gibt, so terminiert auch die normal order Auswertungsreihenfolge

... Church/Rosser (1936)

Insbesondere:

Lazy evaluation "vergleichbar effizient" wie applicative (eager) order, falls alle Argumente benötigt werden.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

423

Frei nach Shakespeare...

Eager or lazy evaluation: that is the question.

Quot capita, tot sensa.

Oder: Die Antworten sind verschieden...

- eager evaluation (z.B. ML, Scheme (abgesehen von Macros)...))
- lazy evaluation (z.B. Haskell, Miranda,...)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

424

Lazy vs. Eager: Eine Abwägung (1)

Lazy Evaluation

- Vorteile
 - terminiert mit Normalform, wenn es eine terminierende Auswertungsreihenfolge gibt
 - wertet Argumente nur aus, wenn nötig
 - elegante Behandlung potentiell unendlicher Datenstrukturen
- Nachteile
 - Konzeptuell und implementierungstechnisch anspruchsvoller
 - * partielle Auswertung von Ausdrücken (Seiteneffekte! Beachte: Letztere nicht in Haskell! in Scheme: Verantwortung beim Programmierer.)
 - * Graphtransformationen
 - * Ein-/Ausgabe
 - Volles Verständnis: Domain-Theorie und λ -Kalkül im Detail

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

425

Lazy vs. Eager: Eine Abwägung (2)

Eager Evaluation

- Vorteile
 - Konzeptuell und implementierungstechnisch einfacher
 - Vom mathematischen Standpunkt oft "natürlicher" (Beispiel: `first (42,infIntSet)`)
 - (Einfache(re) Integration imperativer Konzepte)
- Within.. eager oder lazy – eine Frage des Anwendungsprofils!*

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

426

Zu guter Letzt

Wäre ein Haskell-Compiler (Interpreter) korrekt, der die Fakultätsfunktion applikativ auswertet?

Ja, weil die Funktion *fac strict* in ihrem Argument ist...

↔ eager evaluation oder auch strict evaluation!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

427

Kapitel 10: λ -Kalkül

...formale Fundierung nahezu aller funktionalen Programmiersprachen

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

428

Intuitive vs. formale Berechenbarkeit

Ausgangspunkt...

Intuitiv berechenbar ... "Wenn es eine *irgendwie machbare effektive mechanische Methode* gibt, die zu jedem Argument aus dem Definitionsbereich nach endlich vielen Schritten den Funktionswert konstruiert und die für alle anderen Argumente entweder mit einem speziellen Fehlerwert oder nie abbricht".

Zentrale Frage...

Lasst sich der Begriff "intuitiver Berechenbarkeit" formal fassen?

Zur Beantwortung nötig...

Formale Berechnungsmodelle!

...d.h. *Explikationen* des Begriffs "intuitiver Berechenbarkeit".

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

429

Der λ -Kalkül... (1)

- ...ein spezielles formales Berechnungsmodell, wie viele andere auch, z.B.
 - allgemein rekursive Funktionen (Hebrand 1931, Gödel 1934, Kleene 1936)
 - Turing-Maschinen (Turing 1936)
 - μ -rekursive Funktionen (Kleene 1936)
 - Markov-Algorithmen (Markov 1951)
 - ...
- ...geht zurück auf Alonzo Church (1936)
- ...Berechnungen über Paaren, Listen, Bäumen, auch unendlichen, Funktionen höherer Ordnung einfach ausdrückbar
- ...in diesem Sinne "praxisnäher/realistischer" als andere formale Berechnungsmodelle

Der λ -Kalkül... (2)

Church'sche These

Eine Funktion ist genau dann intuitiv berechenbar, wenn sie λ -definierbar ist (d.h. im λ -Kalkül ausdrückbar ist).

Beweis? ...schlechterdings unmöglich!

Aber...

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

431

Der λ -Kalkül... (3)

Man hat bewiesen...

- Alle der obigen Berechnungsmodelle sind gleich mächtig.

Das kann als Hinweis darauf verstanden werden, dass alle der obigen Berechnungsmodelle den Begriff wahrscheinlich "gut" charakterisieren!

Aber: es schließt nicht aus, dass morgen ein mächtigeres formales Berechnungsmodell gefunden wird, das dann den Begriff der intuitiven Berechenbarkeit "besser" charakterisierte.

Präzedenzfall: Primitiv rekursive Funktionen

- ...bis Ende der 20er-Jahre als adäquate Charakterisierung intuitiver Berechenbarkeit akzeptiert (auch von Hilbert)
- ...tatsächlich jedoch: echt schwächeres Berechnungsmodell
- ...Beweis: Ackermann-Funktion ist berechenbar, aber nicht primitiv rekursiv (Ackermann 1928)

Die Ackermann-Funktion

... "gerühmtberüchtigtes" Beispiel einer

- zweifellos (intuitiv) berechenbaren, aber nicht primitiv rekursiven Funktion!

```
ack :: (Integer,Integer) -> Integer
ack (m,n)
  | m == 0      = n+1
  | (m > 0) && (n == 0) = ack (m-1,1)
  | (m > 0) && (n /= 0) = ack (m-1,ack(m,n-1))
```

~> ...hier in Haskell-Notation!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 433

Der λ -Kalkül... (4)

...ausgezeichnet durch

- *Einfachheit*
...nur wenige syntaktische Konstrukte, einfache Semantik
- *Ausdruckskraft*
... Turing-mächtig, alle "intuitiv berechenbaren" Funktionen im λ -Kalkül ausdrückbar

Darüberhinaus...

~> Bindeglied zwischen funktionalen Hochsprachen und ihren maschinenahnen Implementierungen.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 434

Wir unterscheiden...

- *Reiner λ -Kalkül*
...reduziert auf das "absolut Notwendige"
~> besonders bedeutsam in Untersuchungen zur Theorie der Berechenbarkeit

- *Angewandte λ -Kalküle*
...syntaktisch angereichert, praxisnäher

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 435

Alternativ...

...die Syntax in (modifizierter) Backus-Naur-Form (BNF):

```
e ::=
  ::= x      ( $\lambda$ -Ausdrücke)
  ::=  $\lambda x.e$  (Namen (Identifikatoren))
  ::= e e    (Abstraktion)
  ::= e e    (Applikation)
  ::= (e)
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 437

Angewandte λ -Kalküle

Angewandte λ -Kalküle sind syntaktisch angereichert.

Beispielsweise...

- Auch Konstanten, Funktionsnamen oder "übliche" Operatoren können Namen (im weiteren Sinn) sein
(Bsp.: 1, 3.14, *true*, *false*, +, *, -, fac, simple, ...)

- Ausdrücke können...

– Komplexer sein
(Bsp.: if e then e₁ else e₂ fi ... statt cond e e₁ e₂ für cond geeignete Funktion)

– getypt sein
(Bsp.: 1 : IN, true : Bool, ...)

- ...

λ -Ausdrücke sind dann beispielsweise auch...

- Applikationen: fac 3, simple x y z (entspricht ((simple x) y) z, ...)
- Abstraktionen: $\lambda x.(x + x)$, $\lambda x.\lambda y.\lambda z.(x * (y - z))$, 2 + 3
(λx .if odd x then x*2 else x div 2 fi) 42, ...)

Reiner λ -Kalkül: Syntax

Die Menge *Exp* der Ausdrücke des (reinen) λ -Kalküls, kurz λ -Ausdrücke, ist definiert durch:

- Jeder *Name (Identifier)* ist in *Exp*.
(Bsp.: a, b, c, ..., x, y, z, ...)
- *Abstraktion*: Wenn x ein Name und e aus *Exp* ist, dann ist auch $(\lambda x.e)$ in *Exp*. *Sprechweise*: Funktionsabstraktion mit formalem Parameter x und Rumpf e.
(Bsp.: $(\lambda x.(x x))$, $(\lambda x.\lambda y.\lambda z.(x (y z)))$, ...)
- *Applikation*: Wenn f und e in *Exp* sind, dann ist auch (f e) in *Exp*. *Sprechweisen*: Anwendung von f auf e, f heißt auch *Rator*, e auch *Rand*.
(Bsp.: $((\lambda x.(x x)) y)$, ...)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 436

Konventionen

- Überflüssige Klammern können weggelassen werden.

Dabei gilt:

- *Rechtsassoziativität* für λ -Sequenzen in Abstraktionen
Bsp.: $-\lambda x.\lambda y.\lambda z.(x (y z))$ kurz für $(\lambda x.(\lambda y.(\lambda z.(x (y z))))$,
– $\lambda x.e$ kurz für $(\lambda x.(e))$
- *Linksassoziativität* für Applikationssequenzen
Bsp.: $-\ e_1 e_2 e_3 \dots e_n$ kurz für $(\dots((e_1 e_2) e_3) \dots e_n)$,
– $(e_1 e_2)$ kurz für $e_1 e_2$

- Der Rumpf einer λ -Abstraktion ist der längstmögliche dem Punkt folgende λ -Ausdruck

Bsp.: $-\lambda x.e$ f entspricht $\lambda x.(e f)$, nicht $(\lambda x.e) f$

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 438

In der Folge

...erlauben wir uns die Annehmlichkeit Ausdrücke, für die wir eine eingeführte Schreibweise haben (z.B. *n * fac (n - 1)*) in dieser gewohnten Weise zu schreiben, auch wenn wir die folgenden Ergebnisse für den reinen λ -Kalkül formulieren.

Rechtfertigung...

- Resultate der theoretischen Informatik, insbesondere Alonzo Church. *The Calculi of Lambda-Conversion*. Annals of Mathematical Studies, Vol. 6, Princeton University Press, 1941
- ...zur Modellierung von ganzen Zahlen, Wahrheitswerten, etc. durch geeignete Ausdrücke des reinen λ -Kalküls

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 440

Freie und gebundene Variablen (1)

...in λ -Ausdrücken

Die Menge der *frei* vorkommenden Variablen...

$$\begin{aligned} \text{free}(x) &= \{x\}, \text{ wenn } x \text{ ein Variablenname ist} \\ \text{free}(\lambda x.e) &= \text{free}(e) \setminus \{x\} \\ \text{free}(f e) &= \text{free}(f) \cup \text{free}(e) \end{aligned}$$

Umgekehrt...

Die Menge der *gebunden* vorkommenden Variablen...

$$\begin{aligned} \text{bound}(\lambda x.e) &= \text{bound}(e) \cup \{x\} \\ \text{bound}(f e) &= \text{bound}(f) \cup \text{bound}(e) \end{aligned}$$

Beachte: gebunden \neq nicht frei !
...sonst wäre etwa "x, gebunden in y"

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

442

Freie und gebundene Variablen (2)

Betrachte: $(\lambda x. (x y)) x$

- Gesamtausdruck
 - x kommt frei und gebunden in $(\lambda x. (x y)) x$ vor
 - y kommt frei in $(\lambda x. (x y)) x$ vor
- Teilausdrücke
 - x kommt gebunden in $(\lambda x. (x y))$ vor und frei in $(x y)$ und in x
 - y kommt frei in $(\lambda x. (x y))$, $(x y)$ und y vor

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

442

Gebunden vs. gebunden an...

Wir müssen unterscheiden...

- Eine *Variable* ist *gebunden*...
- Ein *Variablenvorkommen* ist *gebunden an*...

Gebunden und *gebunden an* ... unterschiedliche Konzepte!
Letzteres meint:

- Ein (definierendes oder angewandtes) Variablenvorkommen ist an ein definierendes Variablenvorkommen gebunden

Definition

- *Definierendes* V.vorkommen ...Vorkommen unmittelbar nach einem λ
- *Angewandtes* V.vorkommen ...jedes nicht definierende Vorkommen

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

443

Der λ -Kalkül: Vorwärts zur Semantik

Zentral sind folgende Begriffe...

- (Syntaktische) Substitution
- Konversionsregeln / Reduktionsregeln

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

444

Syntaktische Substitution

Erster zentraler Begriff... (syntaktische) *Substitution*

$$\begin{aligned} x[e/x] &= e, \text{ wenn } x \text{ ein Name ist} \\ y[e/x] &= y, \text{ wenn } y \text{ ein Name mit } x \neq y \text{ ist} \end{aligned}$$

$$(f g)[e/x] = (f[e/x]) (g[e/x])$$

$$(\lambda x.f)[e/x] = \lambda x.f$$

$$(\lambda y.f)[e/x] = \lambda y.(f[e/x]), \text{ wenn } x \neq y \text{ und } y \notin \text{free}(e)$$

$$(\lambda y.f)[e/x] = \lambda z.(((f[y/z])[e/x]), \text{ wenn } x \neq y \text{ und } y \in \text{free}(e), \\ \text{wobei } x \neq z \text{ und } z \notin \text{free}(e) \cup \text{free}(f))$$

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

445

(Syntaktische) Substitution

Einige Beispiele zur Illustration...

- $((x y) (y z)) [a+b/y] = ((x (a+b)) ((a+b) z))$
- $\lambda x. (x y) [a+b/y] = \lambda x. (x (a+b))$
- Aber: $\lambda x. (x y) [a+b/x] = \lambda x. (x y)$
- Achtung: $\lambda x. (x y) [x+b/y] \rightsquigarrow \lambda x. (x (x+b))$
...ohne Umbenennung *Bindungsfehler!*

$$\begin{aligned} \text{Deshalb: } \lambda x. (x y) [x+b/y] &= \lambda z. ((x y)[z/x]) [x+b/y] \\ &= \lambda z. (z y) [x+b/y] \\ &= \lambda z. (z (x+b)) \\ &\dots \text{dank Umbenennung kein Bindungsfehler!} \end{aligned}$$

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

446

Konversionsregeln

Zweiter zentraler Begriff: λ -*Konversionen*...

- α -Konversion (Umbenennung formaler Parameter)
 $\lambda x.e \Leftrightarrow \lambda y.e[y/x]$, wobei $y \notin \text{free}(e)$
 - β -Konversion (Funktionsanwendung)
 $(\lambda x.f) e \Leftrightarrow f[e/x]$
 - η -Konversion (Elimination redundanter Funktion)
 $\lambda x.(e x) \Leftrightarrow e$, wobei $x \notin \text{free}(e)$.
- \rightsquigarrow führen auf eine operationelle Semantik des λ -Kalküls.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

447

Sprechweisen

...im Zusammenhang mit Konversionsregeln

- Von links nach rechts gerichtete Anwendungen der β - und η -Konversion heißen β - und η -Reduktion.
- Von rechts nach links gerichtete Anwendungen der β -Konversion heißen β -Abstraktion.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

448

Intuition hinter den Konversionsregeln

Noch einmal zusammengefasst...

- α -Konversion... erlaubt die konsistente Umbenennung formaler Parameter von λ -Abstraktionen
- β -Konversion... erlaubt die Anwendung einer λ -Abstraktion auf ein Argument (Achtung: Gefahr von Bindungsfehler! Abhilfe: α -Konversion!)
- η -Konversion... erlaubt die Elimination redundanter λ -Abstraktionen

Bsp.: $(\lambda x.\lambda y.x+y) (y*2) \Rightarrow \lambda y.(y*2)+y$
 \rightsquigarrow Bindungsfehler ("y wird eingefangen")

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

449

Beispiel für λ -Reduktion

$(\lambda x.\lambda y.x*y) ((\lambda x.\lambda y.x+y) 9 5) 3$	(β -Reduktion)
$\Rightarrow (\lambda x.\lambda y.x*y) ((\lambda y.9+y) 5) 3$	(β -Reduktion)
$\Rightarrow (\lambda y.((\lambda y.9+y) 5)*y) 3$	(β -Reduktion)
$\Rightarrow (\lambda y.(9+5)*y) 3$	(β -Reduktion)
$\Rightarrow (9+5) * 3$	(β - und η -Reduktion nicht anwendbar)

\rightsquigarrow Weitere Regeln zur Reduktion primitiver Operationen in erweitertem λ -Kalkül (Auswertung arithmetischer Ausdrücke, bedingte Anweisungen, Listenoperationen, ...), sog. δ -Regeln.

\rightsquigarrow solche Erweiterungen sind praktisch notwendig und einsichtig, aber für die Theorie (der Berechenbarkeit) kaum relevant.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

450

Reduktionsfolgen & Normalformen (1)

- Ein λ -Ausdruck ist in *Normalform*, wenn er durch β -Reduktion und η -Reduktion nicht weiter reduzierbar ist.
- (Praktisch relevante) Reduktionsstrategien
 - normal order (leftmost-outermost)
 - applicative order (leftmost-innermost)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

451

Reduktionsfolgen & Normalformen (2)

- *Beachte*: Nicht jeder λ -Ausdruck ist zu einem λ -Ausdruck in Normalform konvertierbar (Eindlosrekursion).
Bsp.: (1) $\lambda x.(x x) \lambda x.(x x) \Rightarrow \lambda x.(x x) \lambda x.(x x) \Rightarrow \dots$
(2) $(\lambda x.42) (\lambda x.(x x) \lambda x.(x x))$ (hat Normalform!)

Zentrale Resultate:

- Wenn ein λ -Ausdruck zu einem λ -Ausdruck in Normalform konvertierbar ist, dann führt jede terminierende Reduktion des λ -Ausdrucks zum (bis auf α -Konversion) selben λ -Ausdruck in Normalform (bis auf α -Konversion).
- Durch Reduktionen im λ -Kalkül sind genau jene Funktionen berechenbar, die Turing-, Markov-, ... berechenbar sind!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

452

Church-Rosser-Theoreme

Seien e_1 und e_2 zwei λ -Ausdrücke...

Theorem 1

Wenn $e_1 \leftrightarrow^* e_2$, dann gibt es einen λ -Ausdruck e mit $e_1 \Rightarrow^* e$ und $e_2 \Rightarrow^* e$ (sog. *Konfluenz Eigenschaft*, *Diamanteigenschaft*)

Informell ...: wenn eine Normalform ex., dann ist sie eindeutig (bis auf α -Konversion)!

Theorem 2

Wenn $e_1 \Rightarrow^* e_2$ und e_2 in Normalform, dann gibt es eine normal order Reduktionsfolge von e_1 nach e_2 (sog. *Standardisierungstheorem*)

Informell ...: normal order Reduktion terminiert am häufigsten!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

453

Semantik von λ -Ausdrücken

- λ -Ausdrücke in Normalform lassen sich (abgesehen von α -Konversionen) nicht weiter vereinfachen (reduzieren)
- Nach dem 1. Church-Rosser-Theorem ist die Normalform eines λ -Ausdrucks eindeutig bestimmt, wenn sie existiert (wieder abgesehen von α -Konversionen)

Das legt folgende Sichtweise nahe...

- Besitzt ein λ -Ausdruck eine Normalform, so ist dies sein Wert.
- *Umgekehrt*: Die *Semantik* (*Bedeutung*) eines λ -Ausdrucks ist seine Normalform, wenn sie existiert, ansonsten ist sie undefiniert.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

454

Rekursion. Und wie sie behandelt wird...

Erinnerung...

fac n = if n == 0 then 1 else (n * fac (n - 1))

oder alternativ:

fac = $\lambda n.$ if n == 0 then 1 else (n * fac (n - 1))

"Problem" ...

λ -Abstraktionen sind *anonymous*.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

455

Lösung: Der Y-Kombinator

Y-Kombinator: $Y = \lambda f.(\lambda x.(f (x x))) \lambda x.(f (x x))$

Es gilt: Für jeden λ -Ausdruck e ist $(Y e)$ zu $(e (Y e))$ konvertierbar:

$$\begin{aligned} Y e &\Rightarrow \lambda x.(e (x x)) \lambda x.(e (x x)) \\ &\Rightarrow e (\lambda x.(e (x x))) \lambda x.(e (x x)) \\ &\Leftrightarrow e (Y e) \end{aligned}$$

Mithilfe des Y-Kombinators lässt sich Rekursion realisieren.

Intuition:

$$\begin{aligned} f &= \dots f \dots \text{ (rekursive Darstellung)} \\ &\rightsquigarrow f = \lambda f.(\dots f \dots) f \text{ (}\lambda\text{-Abstraktion)} \\ &\rightsquigarrow f = Y \lambda f.(\dots f \dots) \text{ (nicht-rekursive Darstellung)} \end{aligned}$$

Bemerkung:
 λ -Terme ohne freie Variablen heißen *Kombinatoren*.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

456

Zur Übung empfohlen

...Anwendung des Y-Kombinators

Betrachte...

```
fac = Y λf (λn. if n == 0 then 1 else n * f (n - 1))
```

Rechne nach...

```
fac 1 ⇒ ... ⇒ 1
```

Überprüfe dabei...

Der Y-Kombinator realisiert Rekursion durch wiederholtes Kopieren

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

457

Praktisch relevant: Typisierte λ-Kalküle

Jedem λ-Ausdruck ist ein Typ zugeordnet

Beispiele:

```
3 :: Integer
(*) :: Integer -> Integer -> Integer
(λx.2 * x) :: Integer -> Integer
(λx.2 * x) 3 :: Integer
```

Einschränkung: Typen müssen konsistent sein (wohltypisiert)

Problem jetzt: Selbstanwendung im Y-Kombinator

~> Y nicht endlich typisierbar!

Abhilfe: explizite Rekursion zum Kalkül hinzufügen mittels Hin- und zurücknahme der Reduktionsregel $Y e \Rightarrow e (Y e)$

...nebenbei: zweckmäßig auch aus Effizienzgründen!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

458

Zurück zu Haskell...

- Haskell beruht auf typisiertem λ-Kalkül
- Übersetzer/Interpreter überprüft, ob alle Typen konsistent sind
- Programmierer kann Typdeklarationen angeben (Sicherheit), muss aber nicht (bequem, manchmal unerwartete Ergebnisse)
- fehlende Typinformation wird vom Übersetzer inferiert (be-rechnet)
- Rekursive Funktionen direkt verwendbar (daher in Haskell kein Y-Kombinator notwendig)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

459

Ergänzende und weiterführende Literaturhinweise

- A. Church. *The Calculi of Lambda-Conversion*. Annals of Mathematical Studies, Vol. 6, Princeton University Press, 1941.
- H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. (Revised Edn.), North Holland, 1984.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

460

Zum Abschluss von Kapitel 10

...λ-artige Funktionsnotation in Haskell

...am Beispiel der Fakultätsfunktion:

```
fac :: Int -> Int
```

```
fac = \n-> (if n == 0 then 1 else (n * fac (n - 1)))
```

Mithin in Haskell: "\ " statt "λ" und "->" statt "λ"

Anekdote (vgl. P. Pepper [4]):

```
(nλ + 1) ~> (λn.n + 1) ~> (λn.n + 1) ~> \n->n + 1
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

461

Kapitel 11: Programmieren im Großen, Module, abstrakte Datentypen, reflexive Programmierung

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

462

Programmieren im Großen

Das Modulkonzept von Haskell...

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

463

Modularisierung im allgemeinen (1)

Intuitiv:

- Zerlegung großer Programm(systeme) in kleinere Einheiten, genannt *Module*

Ziel:

- Sinnvolle, über- und durchschaubare Organisation des Gesamtsystems

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

464

Modularisierung im allgemeinen (2)

Vorteile:

- *Arbeitsphysiologisch* ... Unterstützung arbeitsteiliger Programmierung
- *Softwaretechnisch* ... Unterstützung der Wiederverwendung von Programmen und Programtteilen
- *Implementierungstechnisch* ... Unterstützung von "separate compilation"

Insgesamt:

- Höhere Effizienz der Softwareerstellung bei gleichzeitiger Steigerung der Qualität (Verlässlichkeit) und reduzierten Kosten

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

465

Modularisierung im allgemeinen (3)

Anforderungen an das Modulkonzept zur Erreichung vorgenannter Ziele:

- Unterstützung des Geheimnisprinzips
 - ...durch Trennung von
 - Schnittstelle (Import/Export)
 - ... *wie interagiert das Modul mit seiner Umgebung?*
 - Welche Funktionalität stellt es zur Verfügung (Export), welche Funktionalität erwartet es (Import)?
 - Implementierung (Daten/Funktionen)
 - ... *wie ist die Funktionalität des Moduls realisiert?*
- in einem Modul.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

466

Module in Haskell – Allgemeiner Aufbau

```
module MyModule where

-- Daten- und Typdefinitionen
data D1 ... = ...
...
data Dn ... = ...

type T1 = ...
...
type Tn = ...

-- Funktionsdefinitionen
f1 :: ...
f1 ... = ...
...
fn :: ...
fn ... = ...
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

467

Das Modulkonzept von Haskell

...unterstützt:

- Export
 - Selektiv/Nicht selektiv
 - Import
 - Selektiv/Nicht selektiv
 - Qualifiziert
 - Mit Umbenennung
- ...unterstützt nicht:
- automatischer Reexport!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

468

Import: Nicht selektiv

```
module M1 where
...

module M2 where
import M1 -- Nicht selektiver Import:
-- Alle im Modul M1 (sichtbaren) Bezeichner/
-- Definitionen werden importiert und koennen
-- in M2 benutzt werden.
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

469

Import: Selektiv

```
module M1 where
...
-- wie auf voriger Folie

module M2 where
import M1 (D1, D2 (..), T1, f5)
-- Selektiver Import:
-- Lediglich D1 (ohne Konstruktoren),
-- D2 (einschliesslich Konstruktoren),
-- T1 und f5 werden importiert und
-- koennen in M2 benutzt werden.

module M3 where
import M1 hiding (D1, T2, f1)
-- Selektiver Import:
-- Alle (sichtbaren) Bezeichner/
-- Definitionen mit Ausnahme der
-- explizit genannten werden importiert
-- und koennen in M3 benutzt werden.
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

470

Export: Nicht selektiv

```
module M1 where
data D1 ... = ...
...
data Dn ... = ...
-- Nicht selektiver Export:
-- Alle im Modul M1 (sichtbaren)
-- Bezeichner/Definitionen werden
-- exportiert und koennen von anderen
-- Modulen importiert werden.

type T1 = ...
...
type Tn = ...

f1 :: ...
f1 ... = ...
...
fn :: ...
fn ... = .....
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

471

Export: Selektiv

```
module M1 (D1, D2 (..), T1, f2, f5) where
data D1 ... = ...
...
data Dn ... = ...
-- Selektiver Export:
-- Nur die explizit genannten Bezeichner/
-- Definitionen werden exportiert und
-- koennen von anderen Modulen importiert
-- werden.

type T1 = ...
...
type Tn = ...
-- Unterstützt Geheimnisprinzip!

f1 :: ...
f1 ... = ...
...
fn :: ...
fn ... = .....
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

472

Kein automatischer Reexport (1)

```
module M1 where ...

module M2 where
import M1      -- Nicht selektiver Import:
               -- Alle im Modul M1 (sichtbaren) Bezeichner/
fM2...        -- Definitionen werden importiert und koennen
               -- in M2 benutzt werden.

module M3 where
import M2      -- Alle im Modul M2 (sichtbaren) Bezeichner/
               -- Definitionen werden importiert und koennen
               -- in M3 benutzt werden, nicht jedoch von
               -- M2 aus M1 importierte Namen. Mithin: Kein
               -- automatischer Reexport!
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 473

Kein automatischer Reexport (2)

```
Abhilfe: Expliziter Reexport!

module M4 (D1 (...), f1, f2) where -- Selektiver
import M1                          -- Reexport

module M2 (M1, fM2) where          -- Nicht selektiver
import M1                          -- Reexport
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 474

Sonderfälle: Namenskollisionen, Lokale Namen

- Namenskollisionen
 - Abhilfe/Auflösen: Qualifizierter Import
import qualified M1
- Umbenennen importierter Module
 - Lokale Namen importierter
 - * Module
import MyM1 as M1
 - * Bezeichner
import M1 (f1,f2) renaming (f1 to g1, f2 to g2)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 475

Konventionen und gute Praxis

- Konventionen
 - Pro Datei ein Modul
 - Modul- und Dateiname stimmen überein (abgesehen von der Endung .hs bzw. .lhs im Dateinamen).
 - Alle Deklarationen beginnen in derselben Spalte wie `module`.
- Gute Praxis
 - Module unterstützen *eine* (!) klar abgegrenzte Aufgabenstellung (vollständig) und sind in diesem Sinne in sich abgeschlossen; ansonsten Teilen (Teilungskriterium)
 - Module sind "kurz" (ideal: 2 bis 3 Druckseiten; prinzipiell: "so lang wie nötig, so kurz wie möglich")

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 476

Wiederholung (vgl. Folie 27, Vorlesungsteil 1)

"Verstecken" von in `Prelude.hs` vordefinierten Funktionen...

Ergänze...

```
import Prelude hiding (reverse,tail,zip)
```

...am Anfang des Haskell-Skripts im Anschluss an die Modul-Anweisung (so vorhanden), um `reverse`, `tail` und `zip` zu verbergen.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 477

Das Hauptmodul

Module `main`...

- ...muss in jedem Modulsystem als "top-level" Modul vorkommen und eine Definition namens `main` festlegen.
~> ...ist der in einem übersetzten System bei Ausführung des übersetzten Codes zur Auswertung kommende Ausdruck.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 478

Regeln "guter" Modularisierung (1)

(siehe dazu auch M. Chakravarty, G. Keller, *Einführung in die Programmierung mit Haskell*, Kapitel 10, Pearson Studium, 2004.)

Aus *Modulsicht*:

Module sollen...

- einen klar definierten, auch unabhängig von anderen Modulen verständlichen Zweck besitzen
- nur einer Abstraktion entsprechen
- einfach zu testen sein

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 479

Regeln "guter" Modularisierung (2)

Aus *Gesamtprogrammiersicht*:

Aus Modulen aufgebaute Programme sollen so entworfen sein, dass...

- Auswirkungen von Designentscheidungen (z.B. Einfachheit vs. Effizienz einer Implementierung) auf wenige Module beschränkt sind
- Abhängigkeiten von Hardware oder anderen Programmen auf wenige Module beschränkt sind

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 480

Regeln "guter" Modularisierung (3)

Aus *intra- und intermodularer Sicht*:

Zwei zentrale Konzepte in diesem Zusammenhang sind...

- Intramodular: *Kohäsion*
- Intermodular: *Kopplung*

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

481

Regeln "guter" Modularisierung (4)

Aus *intramodularer Sicht*:

Kohäsion

- Anzustreben sind...
 - *Funktionale Kohäsion* (d.h. Funktionen ähnlicher Funktionalität sollten in einem Modul zusammengefasst sein, z.B. Ein-/Ausgabefunktionen, trigonometrische Funktionen, etc.)
 - *Datenkohäsion* (d.h. Funktionen, die auf den gleichen Datenstrukturen arbeiten, sollten in einem Modul zusammengefasst sein, z.B. Baumanipulationsfunktionen, Listenverarbeitungsfunktionen, etc.)
- Zu vermeiden sind...
 - *Logische Kohäsion* (d.h. unterschiedliche Implementierungen der gleichen Funktionalität sollten in verschiedenen Modulen untergebracht sein, z.B. verschiedene Benutzerschnittstellen eines Systems)
 - *Zufällige Kohäsion* (d.h. Funktionen sind ohne sachlichen Grund, zufällig eben, in einem Modul zusammengefasst)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

482

Regeln "guter" Modularisierung (5)

Aus *intermodularer Sicht*:

Kopplung

- beschäftigt sich mit dem Import-/Exportverhalten von Modulen
 - Anzustreben ist...
 - * *Datenkopplung* (d.h. Funktionen unterschiedlicher Module kommunizieren nur durch Datenaustausch (in funktionalen Sprachen per se gegeben))

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

483

Regeln "guter" Modularisierung (6)

Kennzeichen "guter" Modularisierung:

- *Starke Kohäsion*
d.h. enger Zusammenhang der Definitionen eines Moduls
- *Lockere Kopplung*
d.h. wenige Abhängigkeiten zwischen verschiedenen Modulen, insbesondere weder direkte noch indirekte zirkuläre Abhängigkeiten.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

484

Abstrakte Datentypen, kurz: ADTs (1)

Ziel...

- Kapselung von Daten, Realisierung des Geheimnisprinzips auf Datenebene (engl. *information hiding*)

Implementierungstechnisch zentral...

- Haskell's Modulkonzept, speziell "selektiver Export"

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

485

Abstrakte Datentypen, kurz: ADTs (2)

Hinweise auf drei klassische Literaturstellen:

- John V. Guttag, *Abstract Data Types and the Development of Data Structures*: Communications of the ACM, Vol. 20, No. 6, 396-404, 1977.
- John V. Guttag, J. J. Horning, *The Algebra Specification of Abstract Data Types*: Acta Informatica, Vol. 10, No. 1, 27-52, 1978.
- John V. Guttag, Ellis Horowitz, David R. Musser, *Abstract Data Types and Software Validation*, Communications of the ACM, Vol. 21, No. 12, 1048-1064, 1978.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

486

Abstrakte Datentypen, kurz: ADTs (3)

Die grundlegende Idee am Beispiel des Typs *Schlange*:

Schnittstellen/Signatur:

```
NEW:      -> Queue
ADD:      Queue x Item -> Queue
FRONT:    Queue -> Item
REMOVE:   Queue -> Queue
IS_EMPTY: Queue -> Boolean
```

Axiome/Gesetze:

```
(1) IS_EMPTY(NEW) = true
(2) IS_EMPTY(ADD(q,i)) = false
(3) FRONT(NEW) = error
(4) FRONT(ADD(q,i)) = if IS_EMPTY(q) then i else FRONT(q)
(5) REMOVE(NEW) = error
(6) REMOVE(ADD(q,i)) = if IS_EMPTY(q) then NEW
    else ADD(REMOVE(q),i)
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

487

Bsp.: (Warte-) Schlangen als ADT (1)

Warteschlange...

...eine FIFO (first-in/first-out) Datenstruktur

```
module Queue ( Queue,
  emptyQ,
  isEmptyQ,
  joinQ,
  leaveQ,
  ) where
  -- Kein Konstruktorexport!!!
  emptyQ,
  isEmptyQ,
  joinQ,
  leaveQ,
  -- Queue a -> Bool
  -- a -> Queue a -> Queue a
  -- Queue a -> (a, Queue a)
```

... -- Fortsetzung siehe naechste Folie

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

488

Bsp.: (Warte-) Schlangen als ADT (2)

```
... -- Fortsetzung der vorherigen Folie
data Queue = Qu [a]

emptyQ = Qu []

isEmptyQ (Qu []) = True
isEmptyQ _       = False

joinQ x (Qu xs) = Qu (xs++[x])

leaveQ q@(Qu xs)
  | not (isEmptyQ q) = (head xs, Qu (tail xs))
  | otherwise       = error "Niemand wartet!"
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 489

Bsp.: (Warte-) Schlangen als ADT (3)

Notationelle Spielart des Mustervergleichs...

```
leaveQ q@(Qu xs) -- argument as "q or as (Qu xs)"
Mittels...
```

- q Zugriff auf das Argument als Ganzes
- (Qu xs) Zugriff auf/über die Struktur des Arguments

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 490

Bsp.: (Warte-) Schlangen als ADT (2)

Programmiertechn. Vorteile aus der Benutzung von ADTs...

- *Geheimisprinzip*: Nur die Schnittstelle ist bekannt, die Implementierung bleibt verborgen
 - Schutz der Datenstruktur vor unkontrolliertem oder nicht beabsichtigtem/zugelassenem Zugriff
 - Einfache Austauschbarkeit der zugrundeliegenden Implementierung
 - Arbeitsteilige Programmierung

Beispiel: `emptyQ == Qu []`

...führt in Queue importierenden Modulen zu einem Laufzeitehler! (Die Implementierung und somit der Konstruktor `Qu` sind dort nicht sichtbar.)

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 491

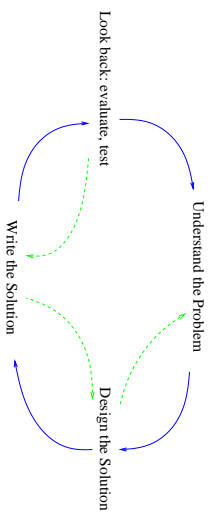
Resümee algebraische vs. abstrakte Datentypen

- Algebraische Datentypen
 - ...werden durch die Angabe ihrer Elemente spezifiziert, aus denen sie bestehen.
- Abstrakte Datentypen
 - ...werden durch ihr Verhalten spezifiziert, d.h. durch die Menge der Operationen, die darauf arbeiten.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 492

Reflektives Programmieren

Der Entwicklungszklus nach S. Thompson, Kap. 11 [3] ...



In jeder dieser Phasen ist es hilfreich, (Sich) Fragen zu stellen!
Für eine beispielhafte Auswahl siehe z. B. S. Thompson, Kap. 11 [3]!

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 493

Kapitel 12: Zusammenfassung und Ausblick

Abschluss und Rückblick...

- Blick über den Gartenzaun ... (ausgewählte) andere funktionale Programmiersprachen
- Rückblick auf die Vorlesung

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 494

Im Rückblick auf die Vorlesung...

- Welche Aspekte funktionaler Programmierung haben wir betrachtet?
 - ...paradigmentypische, sprachunabhängige Aspekte
 - Welche nicht oder nur gestreift?
 - ...sprachabhängige, speziell Haskell-spezifische Aspekte
- Frei nach (und im Sinne von) Dietrich Schwantiz...
- "Alles, was man wissen muss..." um selber weiter zu lernen."*

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 495

Überblick ü. d. Vorlesungsinhalte... (1)

- Vorbesprechung
- Kapitel 1: Einführung und Grundlagen
 - Einführung und Motivation
 - * Warum funktionale Programmierung
 - * Warum Haskell
 - * Einstieg in Haskell und Hugs
- Grundlagen
 - * Elementare Datentypen: Wahrheitswerte, ganze Zahlen, ...
 - * Tupel, Listen und Funktionen, insbesondere notationale Varianten

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008) 496

Überblick ü. d. Vorlesungsinhalte... (2)

- Kapitel 1, Fortführung
 - Funktionen und Funktionssignaturen
 - Funktionen und Funktionsterme, Rekursionstypen, Komplexitätsklassen und Aufrufgraphen, Funktionen und Curryfizierung
- Kapitel 2: Datentypdeklarationen
 - Typsynonyme
 - Algebraische Datentypen
 - * Summentypen
 - * Spezialfälle: Aufzählungs- und Produkttypen
 - * Wichtige Varianten: Rekursive und polymorphe Typen, `newtype`-Deklarationen

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

497

Überblick ü. d. Vorlesungsinhalte... (3)

- Kapitel 3: Polymorphie auf Funktionen und Datentypen
 - Parametrische und ad-hoc Polymorphie
 - Typklassen
 - Vererbung
- Kapitel 4: Listen und Listenkomprehension, Muster
- Kapitel 5: Funktionen höherer Ordnung
- Kapitel 6: Ein- und Ausgabe
- Kapitel 7: Fehlerbehandlung
- Kapitel 8: Monaden
 - Programmierung mit Monaden

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

498

Überblick ü. d. Vorlesungsinhalte... (4)

- Kapitel 9: Ausdrücke, Auswertung von Ausdrücken, Auswertungsstrategien für Funktionen:
 - lazy vs. eager, call by name vs. call by value, normal order vs. applicative order
- Kapitel 10: λ -Kalkül
- Kapitel 11: Programmieren im Großen
 - Das Modulkonzept von Haskell
 - Abstrakte Datentypen
 - Reflektives Programmieren
- Kapitel 12: Zusammenfassung und Ausblick
 - Abschluss und Rückblick
 - Blick über den Gartenzaun und Resümee

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

500

Resümee (2)

Die Fülle an Möglichkeiten (in funktionalen Programmiersprachen) erwächst aus einer kleinen Zahl von elementaren Konstruktionsprinzipien.

P. Pepper [4]

Im Falle von...

- Funktionen
 - (Fkt.-) Applikation, Fallunterscheidung und Rekursion
- Datenstrukturen
 - Produkt- und Summenbildung, Rekursion

Tragen bei zu Mächtigkeit und Eleganz und damit auch zu...

Functional programming is fun!

Rückschauend... war es das?

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

502

ML: Ein "eager" Wettbewerber von Haskell

- ML ist eine strikte funktionale Sprache
- Lexical scoping, Curryfizieren (wie Haskell)
- stark typisiert mit Typinferenz, keine Typklassen
- umfangreiches Typkonzept für Module und ADT
- zahlreiche Erweiterungen (beispielsweise in OCAML) auch für Imperative und objektorientierte Programmierung
- sehr gute theoretische Fundierung

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

503

Resümee (1)

Charakteristika Fkt. und imp. Sprachen (P. Pepper [4])...

- Funktional...
 - Programm ist *Ein-/Ausgaberelation*
 - Programme sind "*zeit*"-los
 - Programmformulierung auf *abstraktem, mathematisch geprägten Niveau*
- Imperativ...
 - Programm ist *Arbeitsanweisung* für eine Maschine
 - Programme sind *zustands-* und "*zeit*"-behaftet
 - Programmformulierung konkret *mit Blick auf eine Maschine*

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

500

Blick über den Gartenzaun

...auf ausgewählte andere funktionale Programmiersprachen:

- ML: Ein "eager" Wettbewerber
 - Lisp: Der Oldtimer
 - APL: Ein Exot
- ...und einige ihrer Charakteristika.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

502

Beispiel: Module/ADTs in ML

```
structure S = struct
  type 't Stack
  val create
  fun push x (Stack xs) = Stack (x::xs);
  fun pop (Stack nil) = Stack nil;
  | pop (Stack (x::xs)) = Stack xs;
  fun top (Stack nil) = nil;
  | top (Stack (x::xs)) = x;
end;

signature st = sig type q; val push: 't -> q -> q; end;
structure S1:st = S;
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

504

Lisp: Der Oldtimer funktionaler Programmiersprachen

- Lisp ist eine noch immer häufig verwendete strikte funktionale Sprache mit imperativen Zusätzen
- umfangreiche Bibliotheken, leicht erweiterbar
- einfache, interpretierte Sprache, dynamisch typisiert
- Listen sind gleichzeitig Daten und Funktionsanwendungen
- nur lesbar, wenn Programme gut strukturiert
- in vielen Bereichen (insbesondere KI, Expertensysteme) erfolgreich eingesetzt
- sehr gut zur Metaprogrammierung geeignet

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

505

Ausdrücke in Lisp

Beispiele für Symbole: `A` (Atom)
`austria` (Atom)
`68000` (Zahl)

Beispiele für Listen: `(plus a b)`
`((meat chicken) water)`
`(unc trw synapse ridge hp)`
`nil` bzw. `()` entsprechen leerer Liste

Eine Zahl repräsentiert ihren Wert direkt — ein Atom ist der Name eines assoziierten Wertes

`(setq x (a b c))` bindet `x` global an `(a b c)`

`(let ((x a) (y b)) e)` bindet `x` lokal in `e` an `a` und `y` an `b`

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

506

Funktionen in Lisp

Erstes Element einer Liste wird normalerweise als Funktion interpretiert, angewandt auf restliche Listenelemente.

`(quote a)` bzw. `'a` liefert Argument `a` selbst als Ergebnis.

Beispiele für primitive Funktionen:

```
(car '(a b c)) => a  
(atom 'a) => t  
(atom '(a)) => nil  
(error) => error  
(cdr '(a b c)) => (b c)  
(eq 'a 'a) => t  
(cdr '(a)) => nil  
(eq 'a 'b) => nil  
(cons 'a '(b c)) => (a b c)  
(cond ((eq 'x 'y) 'b)  
(cons 'a '(b)) => ((a) b)  
(t 'c)) => c
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

507

Definition von Funktionen in Lisp

`(lambda (x y) (plus x y))` ist Funktion mit zwei Parametern
`((lambda (x y) (plus x y)) 2 3)` wendet die Funktion an: $\Rightarrow 5$
`(define (add (lambda (x y) (plus x y))))` definiert einen globalen Namen „add“ für die Funktion

`(defun add (x y) (plus x y))` ist abgekürzte Schreibweise dafür

Beispiel:

```
(defun reverse (l) (rev nil l))  
(defun rev (out in)  
  (cond ((null in) out)  
        (t (rev (cons (car in) out) (cdr in)))))
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

508

Closures

- kein Curryfizieren in Lisp, Closures als Ersatz
- Closures: lokale Bindungen behalten Wert auch nach Verlassen der Funktion
- Beispiel:

```
(let ((x 5))  
  (setf (symbol-function 'test)  
        #'(lambda () x)))
```
- praktisch: Funktion gibt Closure zurück
- Beispiel:

```
(defun create-function (x)  
  (function (lambda (y) (add x y))))
```
- Closures sind flexibel, aber Curryfizieren ist viel einfacher

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

509

Dynamic Scoping — Static Scoping

- lexikalisch: Bindung ortsabhängig (Source-Code)
 - dynamisch: Bindung vom Zeitpunkt abhängig
 - normales Lisp: lexikalisches Binden
 - Beispiel:

```
(setq a 100)  
(defun test () a)  
(let ((a 4)) (test)) => 100
```
 - dynamisches Binden durch `(defvar a)` möglich
- obiges Beispiel liefert damit 4

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

510

Makros

- Code expandiert, nicht als Funktion aufgerufen (wie C)
- Definition: erzeugt Code, der danach evaluiert wird
- Beispiel:

```
(defmacro get-name (x n)  
  (list 'cadr (list 'assoc x n)))
```
- Expansion und Ausführung:

```
(get-name 'a b) => (cadr (assoc 'a b))
```
- nur Expansion:

```
(macroexpand '(get-name 'a b)) => '(cadr (assoc 'a b))
```

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

511

Lisp vs. Haskell: Ein Vergleich

Kriterium	Lisp	Haskell
Basis	einfacher Interpreter	formale Grundlage
Zielsetzung	viele Bereiche	referentiell transparent
Verwendung	noch häufig	zunehmend
Sprachumfang	riesig (kleiner Kern)	moderat, wachsend
Syntax	einfach, verwirrend	modern, Eigenheiten
Interaktivität	hervorragend	nur eingeschränkt
Typisierung	dynamisch, einfach	statisch, modern
Effizienz	relativ gut	relativ gut
Zukunft	noch lange genutzt	einflussreich

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

512

APL: Ein Exot

- APL ist eine ältere applikative (funktionale) Sprache mit imperativen Zusätzen
- zahlreiche Funktionen (höherer Ordnung) sind vordefiniert, Sprache aber nicht einfach erweiterbar
- dynamisch typisiert
- verwendet speziellen Zeichensatz
- Programme sehr kurz und kompakt, aber kaum lesbar
- vor allem für Berechnungen mit Feldern gut geeignet

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

513

Beispiel: Programmmentwicklung in APL

Berechnung der Primzahlen von 1 bis N

Schritt 1. $(\iota N) \circ | (\iota N)$

Schritt 2. $0 = (\iota N) \circ | (\iota N)$

Schritt 3. $+ / | 2] 0 = (\iota N) \circ | (\iota N)$

Schritt 4. $2 = (+ / | 2] 0 = (\iota N) \circ | (\iota N))$

Schritt 5. $(2 = (+ / | 2] 0 = (\iota N) \circ | (\iota N))) / \iota N$

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

514

Erfolgreiche Einsatzfelder funktionaler Programmierung

- Compiler in kompilierter Sprache geschrieben
- Theorembeweiser HOL und Isabelle in ML
- Modelchecker (z.B. Edinburgh Concurrency Workbench)
- Mobility Server von Ericson in Erlang
- Konsistenzprüfung mit Pdfff (Lucent SESS) in ML
- CPL/Kleisli (komplexe Datenbankabfragen) in ML
- Natural Expert (Datenbankabfragen Haskell-ähnlich)
- Ensemble zur Spezifikation effizienter Protokolle (ML)
- Expertensysteme (insbesondere Lisp-basiert)
- ...
- <http://homepages.inf.ed.ac.uk/wadler/realworld/>

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

515

Zum Schluss: Eine hoffnungsvolle Prognose...

The clarity and economy of expression that the language of functional programming permits is often very impressive, and, but for human inertia, functional programming can be expected to have a brilliant future.^(*)

Edsger W. Dijkstra (11.5.1930-6.8.2002)
1972 Recipient of the ACM Turing Award

^(*) Zitat aus: Introducing a course on calculi: Ankündigung einer Lehrveranstaltung an der University of Texas at Austin, 1995.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

516

Lohnenswerte Literaturhinweise...

Bereits in der Vorbesprechung angegeben:

- Wadler, P. *Why no one uses functional languages*. ACM SIGPLAN Notices 33(8), 23-27, 1998
- Aber man sollte stets beide Seiten der Medaille kennen...
- Wadler, P. *An angry half-dozen*. ACM SIGPLAN Notices 33(2), 25-30, 1998
- Hughes, J. *Why Functional Programming Matters*. Computer Journal 32(2), 98-107, 1989
- Philip Wadler. *The Essence of Functional Programming*. In Conference Record of the 19th Annual Symposium on Principles of Programming Languages (POPL'92), eingeladener Vortrag, 1992.
- Simon Peyton-Jones. *Wearing the Hair Shirt: A Retrospective on Haskell*, eingeladener Vortrag, 30th Annual Symposium on Principles of Programming Languages (POPL'03), 2003.

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

517

Organisatorisches LVA-Prüfung 1(3) zur schriftlichen

Hinweise zur schriftlichen LVA-Prüfung...

- Worüber...
 - Vorlesungsstoff, Übungsstoff, und folgender wissenschaftlicher (Übersichts-) Artikel:
Paul Hudak. *Conception, Evolution, and Application of Functional Programming Languages*. ACM Computing Surveys, Vol. 21, No. 3, September 1989, 359 - 411.
- Wann, wo und wie lange...
 - Der Haupttermin ist am...
 - * Do, den 22.01.2009, von 16:00 Uhr s.t. bis 18:00 Uhr, im Rädlinger-Horsaal am Getreidemarkt 2; die Dauer beträgt 90 Minuten.
- Hilfsmittel...
 - Keine.

Organisatorisches LVA-Prüfung 2(3) zur schriftlichen

Hinweise zur schriftlichen LVA-Prüfung...

- Mitzubringen sind...
 - Papier, Stifte, **Studierendenausweis**.
- Anmeldung: Ist erforderlich...
 - *Wann*: Zwischen dem 06.01.2009 und dem 20.01.2009
 - *Wie*: Elektronisch über TLUWIS++
- Voraussetzung...
 - Mindestens 50% der Punkte aus der Laborübung

Funktionale Programmierung WS 2008/2009 (Stand: 06.11.2008)

519

Organisatorisches LVA-Prüfung 3(3) zur schriftlichen

Neben dem Haupttermin wird es drei Nebentermine für die schriftliche LVA-Prüfung geben, und zwar...

- zu Anfang
 - in der Mitte
 - am Ende
- der Vorlesungszeit im SS 2009.
- Danach in jedem Fall Zeugnisausstellung.
- Die genauen Termine werden rechtzeitig auf der Webseite der LVA angekündigt!
- Auch zur Teilnahme an der schriftlichen LVA-Prüfung an einem der Nebentermine ist eine Anmeldung über TLUWIS++ erforderlich.