Zusammenfassung der Programmierparadigmen Vorlesung (Haskell)

ausgearbeitet von
Steve Göring
09.07.2009

Inhalt

1. Vorlesung

1 Definition (Programmier)Paradigma:

= Beispiel/Muster
= Denkmuster, höher geordnetes Prinzip, nachdem man vorgeht
= Kollektion von konzeptionellen Mustern, Vorstellung wie etwas funktioniert bzw. sollte.
= Bereitstellung von Berechnungsmodellen und Denkweise für den Entwurf
= Prinzip welches der Programmiersprache/ Programmiertechnik zu Grunde liegt

2 Paradigmen der höheren Programmiersprachen

Imperativ: Lösung eines Problems wird durch Angabe von Schritten realisiert
Deklarativ: Das Problem wird beschrieben, durch Definitionen und Bedingungen, dadurch kann es gelöst werden.
Applikativ (funktional):
    reine funktionale Sprachen (z.B. Haskell)
    unreine funktionale Sprachen (z.B. Lisp)
Deduktive (logische): z.B. Prolog

3 Funktionales Programmierparadigma

3 1 allgemein

Programm= Funktion von Eingabewerte in Ausgabewerte
Funktion wird aus weiteren Funktionen zusammengesetzt oder ist eine primitive Funktion.
Die Funktion ist das Programm.

3 2 Kategorien der funktionalen Sprachen

nach der Ordnung:
    Erste Ordnung (Funktionen können nur definiert und aufgerufen werden)
    Höhere Ordnung (Funktionen können auch als Parameter oder Ergebnis einer Funktion dienen; Funktionen sind hier auch Werte;
        Wert ist eine Funktion ohne Parameter)

nach der Auswertungsstrategie:
    Strikte Auswertung: (strict evaluation)
        Argumente einer Funktion werden vor dem Eintritt in die Funktion berechnet, wie in z.B. C oder Java
    Bedarfsauswertung: (lazy evaluation, call by need)
        Funktionsargumente werden unausgewertet an die Funktion übergeben, erst wenn sie benötigt werden, werden sie berechnet, und dann nur einmal.
        (außer bei sharing: dort werden gleiche Ausdrücke nicht immer wieder neu berechnet, sondern nur einmal)

nach der Typisierung:
    Statisch Typisiert: Typüberprüfung zur Übersetzungszeit
    Dynamisch Typisiert: Typüberprüfung zur Laufzeit

4 Funktionale Programmierung mit Haskell

4 1 Was ist eine Funktion?

= eine rechtseindeutige Abbildung von einer Menge A (Definitionsbereich) in eine Menge B(Wertebereich)

f: A → B

4 2 Operationen

Sind auch nur Funktionen mit 2 Eingabeparametern.

4 3 Lösung der quadratischen Gleichung

```
-----
x1 p q = -(p/2) + sqrt( (p/2)^2-q)
x2 p q =  -(p/2) - sqrt( (p/2)^2-q)
----
```

Erweiterung mit Test der Diskriminante:

```
d p q = (p/2)^2 -q -- Diskriminante

x1' p q | d p q >=0 = -(p/2) + sqrt( d p q)
-- mit Lokaler Definition:
x1'' p q | dd >=0 = pH + sqrt(dd)
  where dd= d p q
        pH = -(p/2)
```

| = Guard Zeichen= „..unter der Bedingung dass..“

4 4 Typen in Haskell

4 4 1 einfache Typen

ganze Zahlen:	Int (Wortgröße des Prozessors)
	Integer (unbegrenzt)
reele Zahlen:	Float
	Double
Wahrheitswerte:	Bool (false,true)
Zeichen:	Char (z.B. 'T')

Typnamen immer mit Großbuchstaben, Funktionen (Variablen, Parameter) dagegen klein.

4 4 2 zusammengesetzte Typen

Liste: in eckigen Klammern []
Elemente durch Komma getrennt
beliebige Anzahl
nur Elemente gleichen Types!

Tupel: runde Klammern ()
durch Komma getrennt
Anzahl steht fest
Elemente unterschiedlicher Typen möglich

Beispiele:

Liste: [1,2,3] oder (1:2:3:[]) Typ der Liste :: [Int]
: = cons ; [] = leere Liste
Tupel: (4,'A', true) Typ :: (Int,Char;Bool)

4 4 3 Typklassen

Mit :t kann man in Haskell sich den Typ eines Ausdrucks / Funktion anzeigen lassen.

:t [1,2,3] ergibt dann:
[1,2,3] :: Num a => [a]

a bezeichnet hierbei die Typklasse der Zahlen
Eine Typklasse ist eine Menge von Typen. (kein Typ)

4 4 4 Tupel als Ergebnis einer Funktion

Funktionen sind zwar Abbildungen die ein Ergebnis liefern, dieses Ergebnis kann aber auch ein Tupel sein (ein Tupel ist ein Wert). Daher ist es z.B. möglich die Nullstellen einer quadratischen Funktion als Tupel zurückzugeben:

```
d p q = (p/2)^2 -q -- Diskriminante
x12 p q
 | dd >0 = ( ph + sqrt(dd) , ph - sqrt(dd) )
 | dd== 0 = (ph,ph)
 | otherwise = error "komplexe Wurzel"
 where dd= d p q;  ph = -(p/2)
```

liefert beim Aufruf mit : `x12 (-1) (-6)`
`(3.0,-2.0)`

Tupel können auch Parameter einer Funktion sein.

4 5 Funktionstyp

Auch Funktionen haben einen Typ, man kann diesen sogar explizit festlegen.
x1 :: Float -> Float -> Float (von 4 3)

4 6 Auswahlstrukturen mit Haskell

4 6 1 mit Guards

mehrere Guards werden von oben nach unten abgearbeitet, wenn ein Guard true ist dann wird das Ergebnis zurückgegeben

4 6 2 mit If

```
f x = if(x >0 ) then 4 else 4
```

auch Verschachtelung möglich

4 6 3 mit case

```
f x = case compare x 0 of
          GT -> 5
          EQ -> 3
          LT -> 99
```

GT = greater than; EQ = equal ; LT = lower than

2. Vorlesung

1 Tupel

Ist ein Wert, der aus mehren, endlich vielen, eventuell verschiedenen typisierten Werten, den Komponenten besteht.
Die Komponentenanzahl steht fest.

Bsp:

```
(1.7,"HALLO",true) :: ( Float, String, Bool)
```

2 Typendefinitionen

Syntax: type Typname= Typbeschreibung
Bsp:

```
type Student=(Int, String,String )     // (Matrikelnr, Name, Vorname)

type String =[Char]
```

3 Infix /Präfix Funktionen

Funktionen mit 2 Parametern können entweder in der Infix oder Präfix Form definiert werden.

```
Infix: plus a b = a+b
Präfix: a `plus` b = a+b
```

Auch 2 stellige Operatoren können definiert werden, Symbol darf noch nicht verwendet werden:

```
(***):: Float -> Float -> Float
a *** b = a*b
```

4 Mustervergleich bei Funktionen

Ein Tupel kann auch Parameter einer Funktion sein, dadurch ist es möglich mehrere Fälle zu unterscheiden.
Bsp:

```
note:: Student -> Int
note (1,"Max","Muster")=5
note (a,b,c)= 1
```

Aufruf : note (2,"M","M")
liefert: 1

Aufruf : note (1,"Max","Muster")
liefert: 5

Pattern Matching ist eine andere Form der Fallauswahl bei Funktionsdefinitionen.

Falls man einen Parameter nicht in seiner Auswahl benötigt, kann man _ anstelle des Parameternamens verwenden (_= Wildcard , Platzhalter ohne Name)

z.B.:

```
note:: Student -> Int
note (_,"Max",_)=5
note (a,b,c)= 1
```
Alle Studenten mit Vornamen Max bekommen nun eine 5^^

5 Listen

Eine Liste ist ein Wert, der aus keinem oder mehreren gleichtypisierten Werten (Elementen) besteht.
Die Elementeanzahl ist dabei beliebig.

Bsp:
```
[8,3,5.6] :: [Float]
[]::[a]  // Leere Liste hat den Typ [a]
```

5 1 Definition einer Liste

Liste= Leere Liste [] ODER Restliste des Types [a] , wobei man auch den Kopf angeben kann

Doppelpunkt ist dabei der Listenkonstruktor (= Cons)
die Leere Liste ist ein anderer Listenkonstruktor ([])

5 2 Spezielle Listen

5 2 1 String

```
type String=[Char]
```

5 2 2 Arithmetische Sequenz

```
[1..5] = [1,2,3,4,5]
['a','d'..'m'] = "adgjm" , 'd'-'a' = Schrittweite, von a beginnend..
```

5 2 2 List Comprehension

=Beschreibung einer Liste mit Hilfe einer Liste

[(5* x) | x <- ([1..5])]= [5,10,15,20,25]
B A

„quasi“ Für jedes Element aus der Liste A wird der Ausdruck B, über die lokale „Variable“ x , ausgeführt und das Ergebnis kommt in eine neue Liste.

Auch lokale Bindung von „Hilfsvariablen“ möglich, und Bedingungen:

```
[ 5* x | n <- [1..5] , let x= n*2-3 , even n]=   [5,25]
```

5 3 Rekursion über Listen

Basisfall: leere Liste oder eine endlich-elementige Liste
Rekursionsfall: durch Listenmuster (x:xs) „abschlagen des Kopfes“ und Reduktion des Problems

Bsp:

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

laenge [] =0
laenge (x:xs)= 1 + laenge xs
```

5 4 Falten von Listen

```
foldr f k liste
```
Prinzip: Ersetze (:) durch `f` und den Konstruktor [] durch k

Bsp:

```
a `f` b = a+b
k=0
z= foldr f k [1,2,3,4,5]
```

z ermittelt dabei die Summe aller Listeneinträge

[1,2,3,4,5]= 1:2:3:4:5:[]
nach foldr
= 1 `f` 2 `f` 3 `f` 4 `f` 5 `f`0
= 1 + 2 + 3 + 4 + 5 + 0

auch mit Anonymer Funktion möglich

z= foldr (\ x xs -> x+xs) 0 [1,2,3,4,5]

6 Funktionstyp mit Typvariablen

Wenn im Typausdruck Typvariablen auftreten, so nennt man diesen Typausdruck und alle Funktionen die damit arbeiten polymorph

Bei keiner Einschränkung der Typvariablen = universelle Polymorphie
Bei Einschränkungen =nichtuniverselle Polymorphie

7 Erstklassige Werte/höhere Funktionen

Jeder erstklassige Wert kann:
Argument (einer Funktion), Ergebnis (~) oder Komponente einer Datenstruktur sein

Funktion höhere Ordnung kann:
Funktionen als Paramter haben,
Funktion als Ergebnis liefern
Funktionen sind erstklassie Werte.

8 Lambda Abstraktion

= eine anonyme Funktion (=Funktion ohne Funktionsname)

z.B. wenn Funktion ein Paramter ist.

Syntax:

```
lambda::= \ muster [muster] → ausdruck
```

9 Funktionskomposition

= Nacheinanderausführen von Funktionen. g(f(x))

Spezieller Operator (.) , so wird aus
f (f x) = (f . f) x

3. Vorlesung

1 Algebraische Typen in Haskell

1 1 Zweck des Typensystems in Haskell

Typgerechte Verwendung von Ausdrücken
Typ wird zur Übersetzungszeit also statisch geprüft
→ keine Typfehler
Definition eigener Typen
Typen haben Dokumentationscharakter

1 2 Typarten in Haskell

Elementare Typen
Zusammengesetzte Typen

1 3 Konstruktion eigener Typen

Syntax

```
data Typname/konstruktor = Wertkonsturktorfunktionen [deriving]
```

deriving:(optional)
der Typ wir zu einer Instanz der angegebenen Typklasse,erbt somit alle Operationen die für diese Klasse definiert wurden

vorhandene Typklassen (Ableitung nur von diesen möglich):
Eq (== !=),
Ord (vergleichen: >=, <=, <,> setzt Eq voraus) ,
Enum(arithmetische Sequenz) ,
Read ,
Show(Ausgabe),
Bounded

1 3 1 Aufzählтyp

Bsp:

```
data Monate = Jan| Feb|Mae|Apr|Mai|Jun|Jul|Aug|Sep|Okt|Nov|Dez deriving (Eq,Ord)

testaufdezember:: Monate->Bool
testaufdezember Dez=True
testaufdezember _=False
```

1 3 2 Produkttypen

Neuer Typ wird durch Angabe von Funktionen mit anderen Typen definiert
Konstruktorfunktionen mit Parametern

```
data Stift = Fueller Farbe | Kuli Farbe
type Farbe= Int
farbe Fueller farbe = 0
farbe Kuli farbe = 1
```

1 3 3 Polymorphe Typen

= besondere Form der Produkttypen
= Konstruktorfunktionen wird mit Typvariable parametrisiert

1 3 4 Rekursive Typen

Konstruktorfunktionen können den eigenen Typ als Parameter übernehmen
(spezielle Form der Produkttypen)
Bsp: Natürliche Zahlen, Listen, Bäume

1 3 4 1 Natürliche Zahlen

```
data Nat = Zero | Succ Nat deriving Show

nplus a Zero = a
nplus a Succ(b) = Succ ( nplus a b )
```

1 3 4 2 Snoc Liste

= umgekehrte Cons Liste , d.h. Elemente werden rechts angefügt

```
infixl 5 :@:
data Snoc a = II | (Snoc a) :@: a
```
// normale Liste data Snoc a = II | a :@:(Snoc a)

Bsp: ((II :@: 2) :@: 1) :@: 5

1 3 4 3 Bäume

```
data Bt = Lf | Nd Bt Bt
```
Bt= binärer Baum
Lf= Wurzel
Nd = Knoten mit 2 Unterbäumen

1 3 4 4 binäre Blattbäume

```
data BB a= BL a | KN (BB a ) (BB a)
```
Bsp:
b1 = KN(BL 1) (KN (BL 2) (BL 3))

1 3 5 Wert undefined

steht für :
undefinierten Wert
Fehler
nicht beendete Berechnung

Bsp:

```
and false undefined = false
```

2 Polymorphie in Haskell

2 1 Einteilung

universell:

Funktion verhält sich bei allen Typen von Parametern gleich
(dadurch nur eine Funktionsdefinition)
Funktionstyp enthält Typvariable ohne Typeinschränkungen
(parametrische/generische | einschließende/subtyping)

nicht universell (ad hoc):

Funktion verhält sich bei verschiedenen Parametern ähnlich
mehrere Funktionsdefinition für unterschiedliche Typen
(Überladung / overloading | Anpassung / correction)

2 2 Haskell

parametrische Polymorphie (universelle)

Typausdruck der Funktion enthält Typvariablen

Überladung ,implizite/explizite Anpassung (nicht universell)

eingeschränkte Typvariable
durch Typklasse

2 3 Typklassen

2 3 1 allgemein

Typklasse fasst verschiedene Typen zu einer Menge zusammen.
Legt spezielle zulässige Funktionen fest
Menge der Funktionstypen einer Klasse heißt Signatur der Klasse
Elemente der Klasse heißen Instanzen
Typklasse wird groß geschrieben ist aber kein Typ.

2 3 2 Deklaration einer Typklasse

Entweder schon vordefiniert oder selbst definiert oder erweitern.

4. Vorlesung

1 Funktionsanwendung

f x y z wird schrittweise abgearbeitet.
Wende Funktion f an nimmt immer nur einen Parameter

```
f x
f x y
f x y z
```

1 1 Bsp: Typen einer partiellen Funktion

```
f:: String → Int → Bool → Float

f "h"  :: Int → Bool → Float
f "h" 2 :: Bool → Float
f "h" 2 False:: Float
```

1 2 Operatorsektion

Zweistellige Funktionen können auch partiell auf nur ein Argument angewendet werden, die dabei entstandene einstellige Funktion nennt man (Operator-)Sektion.

2 Reduktionsmodelle

2 1 allgemein

Applikative Reduktion (call by value)
Normalform-Reduktion (call by name)
Bedarfsauswertung (call by need)

```
quad x = x*x
```

2 2 Applikative Reduktion

= linkeste innere Reduktion (leftmost innermost); strikte Auswertung

quad (2+3) = erst den Parameter berechnen (2+3=5), = quad(5) = 5*5 = 25

Erst den aktuellen Parameter berechnen und ihn dann für den formalen Parameter einsetzen

von links nach rechts, von innen nach außen

2 3 Normalform Reduktion

=linkeste äußere Reduktion (leftmost outermost); nichtstrikte Auswertung
Wertet Ausdrücke bis zur Kopfnormalform aus.
quad (2+3) = (2+3)*(2+3) = 5*(2+3) = 5*5 = 25
Erst aktuellen Parameter für formalen einsetzen , und dann ausrechnen
von links nach rechts, von außen nach innen, eventuell auch mehrfache Berechnung

2 4 Bedarfsauswertung

=Lazy Evaluation = call by need = call by name + sharing

nur rechnen wenn das Zwischenergebnis auch gebraucht wird
Bsp:

f a b = a

f 1 (99*99) = 1 , (99*99) wird hier nicht gebraucht

3 Referentielle Transparenz und Typbeständigkeit

Referentielle Transparenz= ein Name wird während einer Berechnung einmalig mit einem Wert verbunden
Typbeständigkeit= wenn Typvariable nach Bindung immer den gleichen Typ hat

Substitutionsprinzip:
Gleichwertiges kann durch Gleichwertiges ersetzt werden

4 Beweise

4 1 allgemein

Beweis= logische Begründung, die zeigt das etwas Bestimmtes unter allen Umständen gilt
überzeugt uns, dass unsere Funktionen bestimmte Eigenschaften haben

Gegensatz zum Beweise: Testen

Beweisen und Testen werden bei der Softwareentwicklung benötigt

4 2 Beweis durch „Ausrechnen“

Im Groben durch mathematische Umformungen, oder Funktionsdefinitionen zum Ziel kommen

4 3 Strukturelle Induktion über Listen

Die Aussage A(x:xs) ist zu beweisen.
(IA)Beweise dass A([]) gilt.
(IV) A(xs) gilt
(IS) Mit Hilfe der IV und des Rekursionsschritts beweisen dass A(x:xs) gilt

5 Binäre Bäume (2) in Haskell

5 1 Definition

```
infixl 5 :@
data BB a = BL a | (BB a) :@ (BB a) deriving (Eq,Show)
```

5 2 Parameter des Baumes

5 2 1 Blattanzahl

```
blattanz:: BB b → a
blattanz (BL a)= 1
blattanz (l :@ r )= blattanz l +1 + blattanz r
```

5 2 2 Anzahl innerer Knoten

```
ikanz:: BB b → a
ikanz (BL a)= 0
ikanz (l :@ r )= ikanz l +2 + ikanz r
```

5 3 Binäre Suchbäume- Baumdurchläufe

preorder = Hauptreihenfolge = Element , Linker Teilbaum, Rechter Teilbaum
postorder = Nebenreihenfolge = Linker Teilbaum, Rechter Teilbaum, Element
inorder = symetrische Reihenfolge = Linker Teilbaum, Element, Rechter Teilbaum

6 Lokale Geltungsbereiche

let Ausdruck oder where Klausel

6 1 let

lokale Bindungen werden vor ihrer Verwendung eingefügt
Bsp:

```
h= let u=5.0
       v=2.0
   in (u+v) / (u-v)
```

6 2 where

Bestandteil einer Funktionsdefinition
lokale Bindungen werden nach ihrer Verwendung eingefügt

5. Vorlesung

1 Spezielle Haskellfunktionen

Im Folgenden bezeichnet:
p ein Prädikat (siehe 2); f eine einstellige Funktion ; L eine Liste

1 1 mapWhile

```
mapWhile f p L
```

Pseudoquelltext:

While p=true Do
Begin
 x= erstes Element der Liste L
 packe f(x) in Ergebnisliste
 Lösche erstes Element aus Liste L
End

1 2 filter

```
filter p L
```

wendet das Prädikat p auf die Liste an, falls es true ist, wird das Element in die Ergebnisliste gepackt

Pseudocode:
Foreach Element der Liste L do
 If p = true
   then packe Element in Ergebnisliste

1 3 takeWhile

```
takeWhile p L
```

Pseudocode:
i=0
While p=true do
begin
 packe i-tes Element von L in die Ergebnisliste
end

kopiert im Grunde genommen alle Elemente der Liste L bis das Prädikat p false ergibt.

1 4 iterate

iterate f x
Die Einstellige Funktion f wird wiederholt auf ihren eigenen Funktionswert angewendet, beginnend mit x, die Ergebnisse stehen in einer unendlichen Liste.

2 Prädikat

=eine Funktion mit Parametern die einen Wahrheitswert liefert (bool)
Nach dem Einsetzen der Parameter stellt die rechte Seite der Funktion eine Aussage dar

3 Currying / Uncurrying

curryfizierte Funktion= Funktion mit getrennten Parametern, wenn die Operation immer nur einen Parameter nimmt.

4 Gleichheit von Funktionen

```
mal2 x =x+x
mal2' x = x*2
```

Beide Funktionen beschreiben die gleiche Funktion,

Ein Beweis für die Gleichheit mal2 x = mal2' x $\forall x$ heißt applikativer/punktweiser Beweisstiel (in Haskell möglich)

punktfreier Beweisstiel mal2 = mal2' nicht möglich mit Haskell

5 Rekursionsarten

Rekursive Funktionen lassen sich nur über induktive/rekursive Typen definieren.

5 1 Lineare Rekursion

f [] = Endwert
f (x:xs) = mach was mit dem Kopf x und rufe Rekursiv mit Restliste auf

5 2 Akkumulierende Rekursion

Funktionsparameter sammelt Zwischenergebnisse
Bsp: (Liste umkehren)

```
umkehren ls = hu [] ls
// hu = hilfsfunktion

hu akku []= akku
hu akku (x:xs) = hu (x:akku) xs
```

5 3 Restrekursion (tail recursion)

Spezialfall von linearer Rekursion
keine Auswertung wird aufgeschoben

Beim Aufruf des Basisfalls steht das Ergebnis komplett fest, die vorhergehenden Schritte dienen nur dem Finden,des Basisfalls
Bsp:

```
letztes [] = error "Fehler"
letztes [x]= x
letztes (x:xs)= letztes xs
```

5 4 Gegenseitige Rekursion

2 Funktionen rufen sich gegenseitig Rekursiv auf.
→ ineffizient

5 5 monotone / nichtmonotone Rekursion

monotone Rekursion = ein Wert wird fortlaufend (rekursiv) vergrößert (verkleinert)
nichtmonotone Rekursion= ein Wert wird fortlaufend (rekursiv) vergrößert
oder verkleinert (abwechselnd), Vorgang terminiert aber irgendwann

6 Listenarten in Haskell

6 1 Endliche Liste

Endet mit []
Bsp: 1:2:3:[]

6 2 Partielle Liste

Endet mit undefined und nicht mit []

6 3 Unendliche Listen

Möglich durch Lazy Evaluation (call by need)
Nutzung für:
Modellierung von Vorgängen über die Zeit
Zerlegung größerer Funktionen in kleinere
Unendliche Liste als Grenzwert

Zugriff ist nur auf endliche Teile terminierend möglich
Unendliche Liste von unendlichen Listen ist möglich (List Comprehension)

Bsp:
```
[1..]
```

oder

```
iii :: [Char]
iii= 'i' : iii
```

mit
```
take 10 iii
```
Zugriff auf die Ersten 10 i's

7 Beispiel Algorithmen

7 1 Primzahlen (Sieb des Eratosthenes)

```
primzahlen = map head ( iterate sieb [2..] )
sieb (p:ns) = [ x | x ← ns , x `mod`p /= 0]
```

[2..] unendliche Liste von 2 …
map head = markiert die erste Zahl als Primzahl
sieb = löscht alle vielfachen der ersten Zahl in der Liste und erzeugt neue Liste

7 2 Quadratwurzel nach Newton

```
SQRT x = until done improve x    // Until wiederholt einfach nur bis die
Bedingung wahr ist
 where
          done y = abs(y^2-x)<eps  // Test ob die Näherung erreicht ist
          improve y= (y+x/y)/2.0 // Iterationsschritt
          eps= 0.0000001  / /Genauigkeit
```

7 3 Fibonacci Zahlen

```
fibs :: [Int]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

Info zu zipWith:

```
zipWith (+) [1,2,3] [5,6,7]
```
liefert [6,8,10]

```
zipWith op L1 L2
```
wendet also die 2 stellige Operation op auf die Elemente der Listen L1 und L2 paarweise (also erstes Element von L1 mit erstem von L2 , zweites Element von L1 mit zweitem von L2 …) an.

6. Vorlesung

Thema : Ein und Ausgabe, denke eher dass das „nice to know“ ist

7. Vorlesung

Thema: Beweise

8. Prüfung aus irgendeinem Jahr

8 1 Prüfung

Abschnitt Funktionale Programmierung

Aufgabe 1

Gegeben ist folgende Funktionsdefinition (in Haskell – Syntax)

eins x = "EINS"

Wenn der Ausdruck eins (1 'div' 0) ausgewertet wird, erhält man je nach Auswertungsstrategie (d.h. Bedeutungsstrategie) das Ergebnis „Eins“ oder eine Fehlermeldung. Kreuzen sie in der Tabelle das zutreffende Auswertungsergebnis an, also EINS oder Fehler.

Auswertungsstrategie		EINS	Fehler
applicative order reduction	call by value		
normal order reduction	call by name		
graph reduction	call by need		

Aufgabe 3

Gegeben ist folgende Funktionsdefinition zusammen mit ihrem Typ:

```
fkt :: Int -> String -> Bool -> Char -> Float
fkt :: w x y z = 200.1
```

Welchen Typ hat der Ausdruck (d.h. Typ der partiellen Funktionsanwendung)?

fkt 11 „Juli“ ::

Aufgabe 4

Definieren Sie eine Funktion lookup, die in einer Liste von Paaren nach der ersten Paarkomponente sucht und als Ergebnis eine Liste der zweiten Paarkomponenten von den gefundenen Paaren liefert. Der Typ sei

```
Lookup :: Eq a => a -> [(a,b)] -> [b],
```

wobei der erste Parameter der Suchschlüssel und der zweite Parameter die Paarliste sein soll.

Gegeben ist folgende Funktionsdefinition zusammen mit ihrem Typ:

```
fkt :: Int -> String -> Bool -> Char -> Float
fkt :: w x y z = 200.1
```

Welchen Typ hat der Ausdruck (d.h. Typ der partiellen Funktionsanwendung)?

```
fkt 11 „Juli“ ::
```

Aufgabe 5

Geg. sei der allgemeine Typ des Binärbaums:

```
> data Tr a = Lf | Nd a (Ts a) (Ts a) desiving Show
```

(a) Def. Sie eine Funktion eqt, die überprüft, ob zwei Binärbäume gleich sind, und die den Typ
eqt :: Eq a => (Ts a) -> (Ts a) -> Bool hat.

(b) Wie ist die obige Typdefinition zu ergänzen, damit man sich die Definition von eqt und das Instanzieren für Eq sparen kann?

(c) Machen Sie den Typ (Ts a) zu einer Instanz der Typklasse Eq, explizit
(also nicht wie Antwort b!)

(d) Geben Sie die Typen der beiden (Wert-) Konstruktorfunktionen des Typs (Tr a) an.

Aufgabe 6

Gegeben ist folgender Ausdruck (mit (^) wird Potenzieren angegeben):

```
(take 7.filter (\ x -> 30>x)) [n^2 | n <- [1..10]]
```

Werten Sie diesen Ausdruck aus und geben Sie auch die Zwischenergebnisse an (wenn Sie Teile weglassen, begründen Sie).

```
[n^2 | n <- [1..10]] ==> .....
Filter (\ x -> 30>x)) ... ==> .....
Endergebnis .....
```

Sie bemerken, dass die Länge der Ergebnisliste kl. 7 ist. Welcher Fall in der Definition von *take* beendet die Rekursion bei der obigen Auswertung? Siehe unten in Hilfe – geben Sie die betreffende Nummer aus dem Kommentar an!

Hilfe (aus dem Prelude-File)

```
(.)                 ::      (b -> c) -> (a -> b) -> ( a-> c )
(f.g) x             =       f (g.x)
filter              ::      (a -> Bool) -> [a] -> [a]
filter p xs         =       [x]x <- xs, px]
take                ::      int -> [a] -> [a]
take 0_             =       []                                  --(1)
take _[]            =       []                                  --(2)
take n(x:xs) | n > 0 = x : take(n-1)xs                          --(3)
take _ _            = error “prelude.take: negative argument”
```

Nummer aus dem Kommentar an!

8 2 Lösungen (nur weiter lesen wenn man unbedingt die Lösung haben möchte^^)

Aufgabe 1

Auswertungsstrategie		EINS	Fehler
applicative order reduction	call by value		X
normal order reduction	call by name	X	
graph reduction	call by need	X	

(Punkte : 6)Aufgabe 2
(Punkte : 4)
Variable bezeichnet in einem Geltungsbereich für die gesamte Rechnung einen
konst. Wert -> keine Seiteneffekte (Nebenwirkungen)
Diese Eigenschaft heißt referentielle Transperenz (Bezugstranzperenz).
In der imperativen Programmierung bezeichnet eine Variable einen Behälter-> einen Speicherplatz.

Der Unterschied zwischen funktionaler und imperativer Programmierung
besteht darin, dass es in der fkt. Programmierung die referentielle Tranzparenz gibt.

Das Wesen dieser ist: "Ein Ausdruck bezeichnet in einer bestimmten Umgebung immer den gleichen Wert!"
Folgerung: "Der Wert eines Ausdrucks hängt nur von den Werten seiner Teilausdrücke ab!"

Aufgabe 3

(Punkte : 5)

```
fkt :: Int -> String -> Bool -> Char -> Float
fkt :: w x y z = 200.1

fkt 11 „Juli“ :: -> Bool -> Char -> Float
```

Aufgabe 4

(Punkte : 16)

```
Lookup x [] = []
Lookup x ((z,y):rs)
         | x==z = y : Lookup x rs
         | otherwise = Lookup x rs
```

oder mit if then else

```
Lookup x [] = []
Lookup x ((z,y):rs) = if x==z then  y : Lookup x rs else Lookup x rs
```

(ungetestet)

Aufgabe 5

(Punkte : insgesamt 20)
kein Plan
(a)
(b)
(c)
(d)

Aufgabe 6

(Punkte : 13)
[n^2 | n <- [1..10]] ==> [1, 4, 9, 16,25,36,49,64,81,100] // Quadratzahlen von 1 bis 10
filter (\x -> 30 > x) … ==> [1, 4, 9, 16,25] // Alle Zahlen die kleiner als 30 sind
(\x -> 30 > x) = Prädikat in Lambda Abstraktion
take 7. == nimm die ersten 7 Elemente der Liste, die ist aber nach filter nur 5 elementig, also ist das das Ergebnis

Endergebnis = [1, 4, 9, 16,25]
// mit Haskell getestet

Zeile 2 ist für das Verhalten von take verantwortlich