Funktionale Programmierung

Module, Laziness & Zirkuläre Programme

Andreas Abel und Steffen Jost

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

7. Mai 2012

Aufgaben von Modulsystemen

Größere Softwareprojekte benötigt Struktur $\Longrightarrow$ Module

- Module erlauben Untergliederung in Teilprojekte, welche **separat kompiliert** werden können
- Module organisieren und separieren **Namensraum**
  Modulnamen beginnen mit Großbuchstaben wie Datentypen
- Datei- und Modulnamen müssen in Haskell nicht übereinstimmen, in GHC aber meistens schon
- Haskell Module kennen keine Hierarchie, GHC erlaubt aber Punkte "." im Modulnamen

Module

Ein Modul *Test* das f von *TestAux* importiert und selbst g und f exportiert wird wie folgt definiert:

module *Test*(f, g) **where**
 import *TestAux*(f)
 $g = \ldots$

- Nach dem Modulnamen kann eine Liste von Funktionen und Typen angegeben werden, die aus dem Modul **exportiert** werden. Per default werden alle Definitionen exportiert.
- Im Gegensatz zu SMLs Funktoren gibt es keine parameterisierten Module in Haskell.
- Definitionen anderer Module werden mittels **import** geladen.

Modul Beispiel

Modul **Baum** exportiert und Funktion fringe und Datentyp **Tree** mit Konstruktoren **Leaf** und **Branch** – es müssen nicht immer alle Konstruktoren exportiert werden.

```
module Baum ( Tree(Leaf,Branch), fringe ) where

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

fringe :: Tree a -> [a]
fringe (Leaf x)                = [x]
fringe (Branch left right)     = fringe left ++ fringe righ
```

Namenskonflikte

Mehrfachimporte kein Problem, wenn alle Pfade zum selben Ding führen
Unterschiedliche Dinge gleichen Namens über qualifizierten Import unterscheiden:

```
module Fringe(fringe) where
import Tree(Tree(..))

fringe :: Tree a -> [a]    -- A different definition of f
fringe (Leaf x) = [x]
fringe (Branch x y) = fringe x

module Main where
import Tree ( Tree(Leaf,Branch), fringe )
import qualified Fringe ( fringe )

main = do print (fringe (Branch (Leaf 1) (Leaf 2)))
          print (Fringe.fringe (Branch (Leaf 1) (Leaf 2)
```

Namenskonflikte

Es ist auch möglich, beim Import anderen Qualifizierer zu wählen oder Dinge zu verstecken:

```
import Data.Set as Set
import Data.Map hiding ((!), insert)
import qualified Data.Map ((!), insert)


set   = Set.insert 69 Set.empty
map   = Data.Map.insert 42 empty
```

SML vs. Haskell-Syntax: Module

Modulssystem von Haskell ist etwas schwächer als das von SML (keine Signaturen, keine Funktoren).
Vergelich der Syntax:

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

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

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

```
import Text.Pretty
import qualified System.IO a
```

Auswertungsstrategie

- Eine **Auswertungsstrategie** beschreibt, wie ein Ausdruck ausgewertet wird. Insbesondere:
  1. In welcher Reihenfolge werden die Teilausdrücke bearbeitet?
  2. Werden Funktionsargumente ausgewertet, bevor die Funktion aufgerufen wird?
- Bei Effekten (Wertzuweisung, Ausnahmen, Ein/Ausgabe) ist die Auswertungsstrategie signifikant.
- Bei Programmen ohne Seiteneffektn ist die Auswertereihenfolge egal. (falls das Programm terminiert)

Auswertungsstrategien

```
bar x y z = if x >= 0 then z else x + y
let x = bar (3*4) 1 (1 'div' 0)
```

Call-By-Value

Argumente werden vor dem Funktionsaufruf ausgewertet.

Vorteil: Jedes Argument nur einmal ausgewertet.

Nachteil: Terminiert manchmal nicht.

Im Beispiel wird das Argument `z` ja gar nicht benötigt.

Call-By-Name

Argumente unausgewertet in den Funktionsrumpf eingesetzt.

Vorteil: Unbenutzte Argumente werden gar nicht ausgewertet.

Nachteil: Ineffizient wenn für Funktionen, die ihr Argument mehrfach verwenden.

Bei der Berchnung von `x` wird `3*4` doppelt berechnet.

Lazy Evaluation

Haskell benutzt die Auswertungsstrategie “Lazy Evaluation” (Bedarfsauswertung, verzögerte Auswertung).
Terminierung wie bei Call-By-Name; Effizienz (fast) wie bei Call-By-Value:

- Bei einem Funktionsaufruf wird für jedes Argument ein Zeiger auf den Ausdruck übergeben
- Wird das Argument benötigt, wird es **einmal** ausgewertet und Ausdruck durch den Wert ersetzt. Wird der Zeiger erneut verwendet, ist der Wert sofort vorhanden.

Nur möglich, da die Auswertereihenfolge prinzipiell egal ist.

Vergleich Auswertungsstrategien

Ausgabe eines Programms mit Seiteneffekten, Auswertung von z:

```
import Debug.Trace

foo x y z = y + y + z

z = foo (trace "first" 1)
        (trace "second" 2)
        (trace "third" 3)
```

Call-By-Value: "first second third" 7

Call-By-Name: "second second third" 7

Lazy Evaluation: "second third" 7

Lazy Evaluation: Beispiel

Als Beispiel für Bedarfsauswertung, betrachten wir folgende Funktion:

$$\begin{aligned} \mathit{foo}\ x\ y\ z\ = \ & \textbf{if}\ \underline{x < 0}\ \textbf{then}\ \mathit{abs}\ x \\ & \textbf{else}\ x + y \end{aligned}$$

Auswertungsreihenfolge:

- Die Auswertung des If-Ausdrucks erfordert ein Auswerten von $\underline{x < 0}$ und dieses wiederum ein Auswerten des Arguments x.
- Falls $x < 0$ wahr ist, wird der Wert von $\underline{\mathit{abs}\ x}$ zurückgegeben; weder y noch z werden ausgewertet.
- Falls $x < 0$ falsch ist, wird der Wert von $\underline{x + y}$ zurückgegeben; dies erfordert die Auswertung von y.
- z wird in keinem Fall ausgewertet.
- Insbesondere ist der Ausdruck $\mathit{foo}\ 1\ 2\ (1\ `\mathit{div}`\ 0)$ wohldefiniert.

Lazy Evaluation: Beispiel

Als Beispiel für Bedarfsauswertung, betrachten wir folgende Funktion:

$$\begin{aligned} \mathit{foo}\ x\ y\ z\ = \ & \textbf{if}\ x < 0\ \textbf{then}\ \underline{\mathit{abs}\ x} \\ & \textbf{else}\ x + y \end{aligned}$$

Auswertungsreihenfolge:

- Die Auswertung des If-Ausdrucks erfordert ein Auswerten von $\underline{x < 0}$ und dieses wiederum ein Auswerten des Arguments x.
- Falls $x < 0$ wahr ist, wird der Wert von $\underline{\mathit{abs}\ x}$ zurückgegeben; weder y noch z werden ausgewertet.
- Falls $x < 0$ falsch ist, wird der Wert von $\underline{x + y}$ zurückgegeben; dies erfordert die Auswertung von y.
- z wird in keinem Fall ausgewertet.
- Insbesondere ist der Ausdruck $\mathit{foo}\ 1\ 2\ (1\ `\mathit{div}`\ 0)$ wohldefiniert.

Lazy Evaluation: Beispiel

Als Beispiel für Bedarfsauswertung, betrachten wir folgende Funktion:

$$\begin{aligned} \mathit{foo}\ x\ y\ z\ = \ & \textbf{if}\ x < 0\ \textbf{then}\ \mathit{abs}\ x \\ & \textbf{else}\ \underline{x + y} \end{aligned}$$

Auswertungsreihenfolge:

- Die Auswertung des If-Ausdrucks erfordert ein Auswerten von $\underline{x < 0}$ und dieses wiederum ein Auswerten des Arguments x.
- Falls $x < 0$ wahr ist, wird der Wert von $\underline{\mathit{abs}\ x}$ zurückgegeben; weder y noch z werden ausgewertet.
- Falls $x < 0$ falsch ist, wird der Wert von $\underline{x + y}$ zurückgegeben; dies erfordert die Auswertung von y.
- z wird in keinem Fall ausgewertet.
- Insbesondere ist der Ausdruck $\mathit{foo}\ 1\ 2\ (1\ `\mathit{div}`\ 0)$ wohldefiniert.

Lazy Evaluation: Beispiel

Als Beispiel für Bedarfsauswertung, betrachten wir folgende Funktion:

$$\begin{aligned} \mathit{foo}\ x\ y\ z\ = & \ \textbf{if}\ x < 0\ \textbf{then}\ \mathit{abs}\ x \\ & \ \textbf{else}\ \underline{x + y} \end{aligned}$$

Auswertungsreihenfolge:

- Die Auswertung des If-Ausdrucks erfordert ein Auswerten von $\underline{x < 0}$ und dieses wiederum ein Auswerten des Arguments x.
- Falls $x < 0$ wahr ist, wird der Wert von $\underline{\mathit{abs}\ x}$ zurückgegeben; weder y noch z werden ausgewertet.
- Falls $x < 0$ falsch ist, wird der Wert von $\underline{x + y}$ zurückgegeben; dies erfordert die Auswertung von y.
- z wird in keinem Fall ausgewertet.
- Insbesondere ist der Ausdruck $\mathit{foo}\ 1\ 2\ (1\ `\mathit{div}`\ 0)$ wohldefiniert.

Lazy Evaluation: Beispiel

Als Beispiel für Bedarfsauswertung, betrachten wir folgende Funktion:

$$\begin{aligned} \mathit{foo}\ x\ y\ z\ = \ & \textbf{if}\ x < 0\ \textbf{then}\ \mathit{abs}\ x \\ & \textbf{else}\ \underline{x + y} \end{aligned}$$

Auswertungsreihenfolge:

- Die Auswertung des If-Ausdrucks erfordert ein Auswerten von $\underline{x < 0}$ und dieses wiederum ein Auswerten des Arguments x.
- Falls $x < 0$ wahr ist, wird der Wert von $\underline{\mathit{abs}\ x}$ zurückgegeben; weder y noch z werden ausgewertet.
- Falls $x < 0$ falsch ist, wird der Wert von $\underline{x + y}$ zurückgegeben; dies erfordert die Auswertung von y.
- z wird in keinem Fall ausgewertet.
- Insbesondere ist der Ausdruck $\mathit{foo}\ 1\ 2\ (1\ `\mathit{div}`\ 0)$ wohldefiniert.

Lazy Evaluation

```
g x y =   let  z = x `div` y
          in   if x > y * y then x else z

t2 = g 5 0
```

- `t2` liefert `5`. Da der Wert von `z` nicht benötigt wird, kommt es nicht zur Division durch Null.
- Nicht nur Funktionsargumente werden verzögert ausgewertet, sondern **beliebige Teilausdrücke**.
- **Sequentialität** der Auswertung (erst das, dann das) nicht gegeben, was sehr verwirrend sein kann.

Potentiell unendliche Datenstrukturen

Danke Lazy Evaluation ist es möglich unendliche Datenstrukturen zu definieren:

```
ones  =  1 : ones
twos  =  map (1+) ones

iterate :: (a -> a) -> a -> [a]
iterate f x  =  x : iterate f (f x)

nums = iterate (1+) 0
```

Es wird immer nur soviel von der Datenstruktur ausgewertet wie benötigt wird:

```
> take 10 nums
[0,1,2,3,4,5,6,7,8,9]
```

$\Rightarrow$ Kontrollfluss unabhängig von Daten!

Lazy Evaluation: Beispiel

Hier ist die Auswertungshistorie für *nums*!!2:

Auswertung

nums!!2	$\Longrightarrow$	**Ist Liste leer?**
(*iterate*(1+)0)!!2	$\Longrightarrow$	
(0 : (*iterate*(1+)(1 + 0)))!!2	$\Longrightarrow$	**Ist Index 0?**
(*iterate*(1+)(1 + 0))!!1	$\Longrightarrow$	**Ist Liste leer?**
((1 + 0) : (*iterate*(1+)(1 + (1 + 0))))!!1	$\Longrightarrow$	**Ist Index 0?**
(*iterate*(1+)(1 + (1 + 0)))!!0	$\Longrightarrow$	**Ist Liste leer?**
((1 + (1 + 0)) : (*iterate*(1+)(...))!!0	$\Longrightarrow$	**Ist Index 0?**
(1 + (1 + 0))	$\Longrightarrow$	
(1 + 1)	$\Longrightarrow$	
2		

Weitere Beispiele in Vorlesung "zirkuläre Datenstrukturen"

Lazy Evaluation: Beispiel

Hier ist die Auswertungshistorie für *nums*!!2:

Auswertung

nums!!2	$\Longrightarrow$	**Ist Liste leer?**
(*iterate*(1+)0)!!2	$\Longrightarrow$	
$(0 : (iterate(1+)(1+0)))!!2$	$\Longrightarrow$	**Ist Index 0?**
$(iterate(1+)(1+0))!!1$	$\Longrightarrow$	**Ist Liste leer?**
$((1+0) : (iterate(1+)(1+(1+0))))!!1$	$\Longrightarrow$	**Ist Index 0?**
$(iterate(1+)(1+(1+0)))!!0$	$\Longrightarrow$	**Ist Liste leer?**
$((1+(1+0)) : (iterate(1+)(\ldots))!!0$	$\Longrightarrow$	**Ist Index 0?**
$(1+(1+0))$	$\Longrightarrow$	
$(1+1)$	$\Longrightarrow$	
2		

Weitere Beispiele in Vorlesung "zirkuläre Datenstrukturen"

Lazy Evaluation: Beispiel

Hier ist die Auswertungshistorie für $nums!!2$:

Auswertung

$nums!!2$	$\Longrightarrow$	**Ist Liste leer?**
$(iterate(1+)0)!!2$	$\Longrightarrow$	
$(0 : (iterate(1+)(1+0)))!!2$	$\Longrightarrow$	**Ist Index 0?**
$(iterate(1+)(1+0))!!1$	$\Longrightarrow$	**Ist Liste leer?**
$((1+0) : (iterate(1+)(1+(1+0))))!!1$	$\Longrightarrow$	**Ist Index 0?**
$(iterate(1+)(1+(1+0)))!!0$	$\Longrightarrow$	**Ist Liste leer?**
$((1+(1+0)) : (iterate(1+)(\ldots))!!0$	$\Longrightarrow$	**Ist Index 0?**
$(1+(1+0))$	$\Longrightarrow$	
$(1+1)$	$\Longrightarrow$	
2		

Weitere Beispiele in Vorlesung "zirkuläre Datenstrukturen"

Lazy Evaluation: Beispiel

Hier ist die Auswertungshistorie für $nums!!2$:

Auswertung

$nums!!2$	$\Longrightarrow$	**Ist Liste leer?**
$(iterate(1+)0)!!2$	$\Longrightarrow$	
$(0 : (iterate(1+)(1+0)))!!2$	$\Longrightarrow$	**Ist Index 0?**
$(iterate(1+)(1+0))!!1$	$\Longrightarrow$	**Ist Liste leer?**
$((1+0) : (iterate(1+)(1+(1+0))))!!1$	$\Longrightarrow$	**Ist Index 0?**
$(iterate(1+)(1+(1+0)))!!0$	$\Longrightarrow$	**Ist Liste leer?**
$((1+(1+0)) : (iterate(1+)(\ldots))!!0$	$\Longrightarrow$	**Ist Index 0?**
$(1+(1+0))$	$\Longrightarrow$	
$(1+1)$	$\Longrightarrow$	
2		

Weitere Beispiele in Vorlesung "zirkuläre Datenstrukturen"

Lazy Evaluation: Beispiel

Hier ist die Auswertungshistorie für *nums*!!2:

Auswertung

nums!!2	$\Longrightarrow$	**Ist Liste leer?**
(*iterate*(1+)0)!!2	$\Longrightarrow$	
(0 : (*iterate*(1+)(1 + 0)))!!2	$\Longrightarrow$	**Ist Index 0?**
(*iterate*(1+)(1 + 0))!!1	$\Longrightarrow$	**Ist Liste leer?**
((1 + 0) : (*iterate*(1+)(1 + (1 + 0))))!!1	$\Longrightarrow$	**Ist Index 0?**
(*iterate*(1+)(1 + (1 + 0)))!!0	$\Longrightarrow$	**Ist Liste leer?**
((1 + (1 + 0)) : (*iterate*(1+)(...))!!0	$\Longrightarrow$	**Ist Index 0?**
(1 + (1 + 0))	$\Longrightarrow$	
(1 + 1)	$\Longrightarrow$	
2		

Weitere Beispiele in Vorlesung "zirkuläre Datenstrukturen"

Lazy Evaluation: Beispiel

Hier ist die Auswertungshistorie für $nums!!2$:

Auswertung

$nums!!2$	$\Longrightarrow$	**Ist Liste leer?**
$(iterate(1+)0)!!2$	$\Longrightarrow$	
$(0 : (iterate(1+)(1+0)))!!2$	$\Longrightarrow$	**Ist Index 0?**
$(iterate(1+)(1+0))!!1$	$\Longrightarrow$	**Ist Liste leer?**
$((1+0) : (iterate(1+)(1+(1+0))))!!1$	$\Longrightarrow$	**Ist Index 0?**
$(iterate(1+)(1+(1+0)))!!0$	$\Longrightarrow$	**Ist Liste leer?**
$((1+(1+0)) : (iterate(1+)(\ldots))!!0$	$\Longrightarrow$	**Ist Index 0?**
$(1+(1+0))$	$\Longrightarrow$	
$(1+1)$	$\Longrightarrow$	
2		

Weitere Beispiele in Vorlesung "zirkuläre Datenstrukturen"

Lazy Evaluation: Beispiel

Hier ist die Auswertungshistorie für $nums!!2$:

Auswertung

$nums!!2$	$\Longrightarrow$	**Ist Liste leer?**
$(iterate(1+)0)!!2$	$\Longrightarrow$	
$(0 : (iterate(1+)(1+0)))!!2$	$\Longrightarrow$	**Ist Index 0?**
$(iterate(1+)(1+0))!!1$	$\Longrightarrow$	**Ist Liste leer?**
$((1+0) : (iterate(1+)(1+(1+0))))!!1$	$\Longrightarrow$	**Ist Index 0?**
$(iterate(1+)(1+(1+0)))!!0$	$\Longrightarrow$	**Ist Liste leer?**
$((1+(1+0)) : (iterate(1+)(\ldots))!!0$	$\Longrightarrow$	**Ist Index 0?**
$(1+(1+0))$	$\Longrightarrow$	
$(1+1)$	$\Longrightarrow$	
2		

Weitere Beispiele in Vorlesung “zirkuläre Datenstrukturen”

Lazy Evaluation: Beispiel

Hier ist die Auswertungshistorie für $nums!!2$:

Auswertung

$nums!!2$	$\Longrightarrow$	**Ist Liste leer?**
$(iterate(1+)0)!!2$	$\Longrightarrow$	
$(0 : (iterate(1+)(1+0)))!!2$	$\Longrightarrow$	**Ist Index 0?**
$(iterate(1+)(1+0))!!1$	$\Longrightarrow$	**Ist Liste leer?**
$((1+0) : (iterate(1+)(1+(1+0))))!!1$	$\Longrightarrow$	**Ist Index 0?**
$(iterate(1+)(1+(1+0)))!!0$	$\Longrightarrow$	**Ist Liste leer?**
$((1+(1+0)) : (iterate(1+)(\ldots))!!0$	$\Longrightarrow$	**Ist Index 0?**
$(1+(1+0))$	$\Longrightarrow$	
$(1+1)$	$\Longrightarrow$	
2		

Weitere Beispiele in Vorlesung “zirkuläre Datenstrukturen”

Lazy Evaluation: Beispiel

Hier ist die Auswertungshistorie für $nums!!2$:

Auswertung

$nums!!2$	$\Longrightarrow$	**Ist Liste leer?**
$(iterate(1+)0)!!2$	$\Longrightarrow$	
$(0 : (iterate(1+)(1+0)))!!2$	$\Longrightarrow$	**Ist Index 0?**
$(iterate(1+)(1+0))!!1$	$\Longrightarrow$	**Ist Liste leer?**
$((1+0) : (iterate(1+)(1+(1+0))))!!1$	$\Longrightarrow$	**Ist Index 0?**
$(iterate(1+)(1+(1+0)))!!0$	$\Longrightarrow$	**Ist Liste leer?**
$((1+(1+0)) : (iterate(1+)(\ldots))!!0$	$\Longrightarrow$	**Ist Index 0?**
$(1+(1+0))$	$\Longrightarrow$	
$(1+1)$	$\Longrightarrow$	
2		

Weitere Beispiele in Vorlesung “zirkuläre Datenstrukturen”

Lazy Evaluation: Beispiel

Hier ist die Auswertungshistorie für $nums!!2$:

Auswertung

$nums!!2$	$\Longrightarrow$	**Ist Liste leer?**
$(iterate(1+)0)!!2$	$\Longrightarrow$	
$(0 : (iterate(1+)(1+0)))!!2$	$\Longrightarrow$	**Ist Index 0?**
$(iterate(1+)(1+0))!!1$	$\Longrightarrow$	**Ist Liste leer?**
$((1+0) : (iterate(1+)(1+(1+0))))!!1$	$\Longrightarrow$	**Ist Index 0?**
$(iterate(1+)(1+(1+0)))!!0$	$\Longrightarrow$	**Ist Liste leer?**
$((1+(1+0)) : (iterate(1+)(\ldots))!!0$	$\Longrightarrow$	**Ist Index 0?**
$(1+(1+0))$	$\Longrightarrow$	
$(1+1)$	$\Longrightarrow$	
2		

Weitere Beispiele in Vorlesung “zirkuläre Datenstrukturen”

Beispiel: Sieb des Erathostenes

Unendliche Liste aller Primzahlen:

```
primes :: [Integer]
primes   =   sieve [2..]

sieve :: [Int] -> [Int]
sieve (p:xs)  =  p: sieve [x | x <-xs, x`mod`p /=0]
```

- Wir müssen uns nur um die Daten kümmern, also wie wir die Primzahlen berechnen.
- Die Kontrolle darüber, wie viele Primzahlen wir benötigen, erfolgt später.

Zusammenfassung

- Module erlauben Aufteilung des Codes in separate Einheiten
- Jedes Modul hat seinen eigenen Namensraum
- Module können separat kompiliert werden
- Die Auswertungsstrategie von Haskell ist “Lazy Evaluation”
- Lazy Evaluation: Maximal einmal auswerten, wenn überhaupt
- Lazy Evaluation erlaubt potentiell unendliche Datenstrukturen
- Lazy Evaluation erlaubt Trennung von Daten und Kontrollfluss