Programmierung und Modellierung mit Haskell

Verzögerte Auswertung & Striktheit

Martin Hofmann Steffen Jost

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

30. Juni 2014

Substitutionsmodell Wdh.

Das Substitutionsmodell erklärt das Verhalten von rein funktionalen Sprachen, solange keine Fehler auftreten (und das Programm terminiert).

- Man wählt einen beliebigen Teilausdruck aus und wertet diesen gemäß den geltenden Rechenregeln aus.
- Ein Funktionsaufruf wird durch seinen definierenden Rumpf ersetzt. Dabei werden die formalen Parameter im Rumpf durch die Argumentausdrücke ersetzt ("substitutiert").
- Dies wiederholt man, bis keine auswertbaren Teilausdrücke mehr vorhanden sind.

Es gibt verschiedene **unterschiedliche Strategien**, den als nächstes auszuwertenden Teilausdruck auszuwählen.

Beispiel

```
succ x = x + 1
bar x y z = if 1 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`:

```
bar 1 2 (succ 3) ⇝ bar 1 2 4 ⇝
  if 1 < succ 1 then 4 else 1+2 ⇝
  if 1 < 2 then 4 else 1+2 ⇝ if True then 4 else 1+2 ⇝
  if True then 4 else 3 ⇝ 4
```

Es wäre aber auch möglich so auszuwerten:

```
bar 1 2 (succ 3) ⇝ if 1 < succ 1 then succ 3 else 1+2 ⇝
  if 1 < 2 then succ 3 else 1+2 ⇝
  if True then succ 3 else 1+2 ⇝ succ 3 ⇝ 4
```

Bemerke: Anzahl Auswerteschritte kann sich unterscheiden!

Reduzierbare Teilausdrücke

Ein **Redex** eines Programmausdrucks ist ein **Teilausdruck** davon, welcher weiter ausgewertet werden kann.

Beispiel

$$\texttt{if True then 4 else } \overline{1+2}$$

Dieser Ausdruck enthält zwei Redexe:

1. Fallunterscheidung (Konditional) ausführen
2. Numerische Rechnung ausführen

Verschachtelte Redexe

Im Beispiel ist der Redex der numerischen Rechnung selbst ein Teilausdruck des Konditional-Redex. Wir unterscheiden:

- **Innerster Redex** enthält keinen weiteren Redex als Teilausdruck
- **Äußerster Redex** ist in keinem anderen Redex enthalten.

Der Begriff "Redex" kommt von "reduzieren", auch wenn die Auswertung machmal einen Ausdruck größer machen kann!

Reduzierbare Teilausdrücke (2)

Verschachtelte Redexe bieten manchmal eine Wahl, z.B. Ausdruck `if 1>2 then 4+1 else 5*2` hat drei Redexe

Der Ausdruck `if 1>2 then 4 else 5` hat aber nur einen Redex, da Konditional erst auswertbar ist, wenn Bedingung ein Wert ist!

Keine Reduktion unter einem Lambda

Wir verbieten Reduktion innerhalb eines Funktionsrumpfes, z.B. der Ausdruck `\x -> 1 + 2` enthält keinen Redex.

Hinweis: Reduktion in Funktionsrümpfen kann durchaus erlaubt werden. Wir vereinfachen hier, weil Haskell es auch so macht. Philosophie: Funktionen kann man nur anwenden, aber nicht hineinschauen, d.h. Rumpf muss erst eingesetzt werden.

Auswertungsstrategie

Eine **Auswertungsstrategie** beschreibt, wie ein Programmausdruck auszuwerten ist.

1. In welcher Reihenfolge werden die Teilausdrücke bearbeitet?
2. Werden Funktionsargumente ausgewertet, bevor diese in den Funktionsrumpf eingesetzt werden?

Achtung

- Bei Effekten (Wertzuweisung, Ausnahmen, Ein/Ausgabe) ist die Auswertungsstrategie signifikant.

- Bei Programmen ohne Seiteneffekten ist die Auswertereihenfolge egal falls das Programm terminiert

Seiteneffekte vs. Auswertereihenfolge

In imperativen Sprachen ist die Auswertereihenfolge wichtig.

Beispiel

Der imperative Programmausdruck der Zuweisung x`:=`e
weist als Seiteneffekt der Variablen x den Wert des Ausdrucks e zu,
und wertet selbst zu diesem Wert aus.

Am Anfang gelte `x`=0.

$$\underline{\texttt{x}} \texttt{ + (x:=1)} \leadsto \texttt{0 + } \underline{\texttt{(x:=1)}} \leadsto \texttt{0} \underline{+} \texttt{ 1} \leadsto \texttt{1}$$

Alternative Auswertereihenfolge führt zu anderen Ergebnis:

$$\texttt{x + } \underline{\texttt{(x:=1)}} \leadsto \underline{\texttt{x}} \texttt{ + 1} \leadsto \texttt{1} \underline{+} \texttt{ 1} \leadsto \texttt{2}$$

$\Rightarrow$ Neben dem **was** man berechnen will, muss man in imperativen Sprachen darauf achten, **wie** ausgewertet wird!

Monaden vs. Auswertereihenfolge

Fragen

- Wenn in Haskell die Auswertereihenfolge generell egal ist, wieso funktionieren dann Monaden?
- Warum beachtet die DO-Notation die Auswertereihenfolge?

Antwort

Erfolgt ganz automatisch wegen des Durchfädeln des Kontexts!

Zum Beispiel kann ein if-then-else, dessen Bedingung von dem Kontext der Monade abhängt, erst dann ausgewertet werden, wenn diese Bedingung und damit der Kontext ausgewertet wurden.

Die Auswertereihenfolge für Haskell bleibt also weiterhin beliebig, egal wie, es kommt immer der gleiche Wert heraus!

Termination vs. Auswertereihenfolge

Egal wie ausgewertet wird, Haskell liefert den gleichen **Wert**
Manchmal kommt aber gar kein Wert heraus.

Beispiel

```
bar x y z = if 1 < succ x then z else x+y
```

```
bar 0 2 (3 `div` 0) ⇝ *** Exception: divide by zero
```

Es wäre aber auch möglich so auszuwerten:

```
bar 0 2 (3 `div` 0) ⇝
  if 1 < succ 0 then (3 `div` 0) else 0+2 ⇝
  if 1 < 1 then (3 `div` 0) else 0+2 ⇝
  if False then (3 `div` 0) else 0+2 ⇝ 0+2 ⇝ 2
```

Call-By-Name

Die Auswertestrategie **Call-By-Name** ist festgelegt durch:

- Argumente unausgewertet in den Funktionsrumpf einsetzen.
- Nicht in den Zweigen einer Fallunterscheidung oder eines Pattern-Match reduzieren.

Außerdem wird in Funktionsrümpfen nicht reduziert solange die Argumentet nicht eingesetzt wurden ("Nie unter einem Lambda reduzieren").

Vorteil Unbenutzte Argumente werden gar nicht ausgewertet
Call-By-Name terminiert fehlerfrei, wenn irgendeine andere Auswertestrategie das auch tut.

Nachteil Ineffizient, falls Argumente mehrfach benötigt werden

Beispiel

```
(\x -> \y -> (y,y)) (1 `div` 0) (succ 1) ⇝
  (\y -> (y,y)) (succ 1) ⇝
  (succ 1, succ 1) ⇝ (2, succ 1) ⇝ (2,2)
```

Call-By-Value

Die Auswertestrategie **Call-By-Value** ist festgelegt durch:

- Nur vollständig ausgewertete Argumente werden in Funktionsrümpfe eingesetzt.

Auch bei call-by-value gilt nach wie vor:

- Nicht in den Zweigen einer Fallunterscheidung oder eines Pattern-Match reduzieren.
- Nicht unter einem Lambda reduzieren.

Vorteil Jedes Argument wird nur einmal ausgewertet.

Nachteil Es können fehlerhafte oder nichtterminierende Teilterme ausgewertet werden, obwohl deren Ergebnis gar nicht gebraucht wird.

Beispiel

```
(\x -> \y -> (y,y)) (1 `div` 0) (succ 1)
  ⇝ *** Exception: divide by zero
```

Lazy Evaluation

Haskell benutzt im Prinzip die Strategie "Call-By-Name", verwendet aber die Optimierung **Lazy Evaluation** (Bedarfsauswertung, verzögerte Auswertung).

- Terminierung wie bei Call-By-Name
- Effizienz (fast) wie bei Call-By-Value:

Idee

- Anstelle eines Redex wird ein Verweis eingesetzt
- Wird ein Ausdruck benötigt, wird dieser **einmal** ausgewertet
- Wird der Verweis erneut verwendet, ist der Wert sofort vorhanden.

Nur möglich, da die Auswertereihenfolge prinzipiell egal ist!

Nachteile

- Höherer Speicherverbrauch, wegen Merken der Verweise und Teilausdrücke
- Keine **Sequentialität** der Auswertung (erst das, dann das), was verwirrend sein kann

Vergleich Auswertungsstrategien

Ausgabe eines Programms mit Seiteneffekten, Auswertung von z:

```
import Debug.Trace      -- Von Verwendung wird abgeraten!!
trace :: String -> a -> a

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

Beispiel für Bedarfsauswertung:

```
foo x y z  = if x<0 then abs x else x+y
```

Auswertungsreihenfolge:

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

Potentiell unendliche Datenstrukturen

Lazy Evaluation ermöglicht unendliche Datenstrukturen:

```
ones  =  1 : ones        -- ``unendlich lange'' Liste von 1e
twos  =  map (1+) ones


nums = iterate (1+) 0

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

take :: Int -> [a] -> [a]
take n (x:xs) | n > 0 =  x : take (n-1) xs
take _   _            =  []
```

Potentiell unendliche Datenstrukturen

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

```
nums = iterate (1+) 0
```

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

$\Rightarrow$ Kontrollfluss unabhängig von Daten!

Lazy Evaluation: Beispiel

Auswertung für `nums!!2`:

$\rightsquigarrow$

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		

Beispiel: Sieb des Erathostenes

Unendliche Liste aller Primzahlen:

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

sieve :: [Integer] -> [Integer]
sieve (p:xs) = p : sieve (xs `minus` [p,p+p..])

minus xs@(x:xt) ys@(y:yt) | LT == cmp = x : minus xt ys
                          | EQ == cmp =     minus xt yt
                          | GT == cmp =     minus xs yt
                         where cmp = compare x y
```

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

Striktheit

Eine Argument einer Funktion heißt **strikt**, wenn es auf jeden Fall ausgewertet wird, gleich welchen Wert die anderen Argumente haben.
Beispiel:

```
bar x y z = if 1 < succ x then z else x+y
```

Hier ist x strikt, y und z aber nicht.

Speicherverbrauch Lazy Evaluation

Beispiel

```
sumWithL :: [Int] -> Int -> Int
sumWithL   []  acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h

sumWithL [2,3,4]    1           ⇝
sumWithL   [3,4]   (1+2)        ⇝
sumWithL     [4]  ((1+2)+3)     ⇝
sumWithL      [] (((1+2)+3)+4)  ⇝
                  ((1+2)+3)+4   ⇝
                    ((3)+3)+4   ⇝ (6)+4 ⇝ 10
```

Obwohl `sumWithL` endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹

Striktheit erzwingen

Haskell bietet daher einen Mechanismus, welcher die **strikte Auswertung** erzwingt:

```
($!) :: (a -> b) -> a -> b
```

Der Ausdruck `f $! x` erzwingt die Auswertung von `x` vor der Anwendung von `f`.

Beispiel:

```
sumWithS :: [Int] -> Int -> Int
sumWithS   []  acc = acc
sumWithS (h:t) acc = sumWithS t $! acc+h

sumWithS [2,3,4]     1    ⇝
sumWithS   [3,4] $! (1+2) ⇝ sumWithS [3,4] 3
sumWithS     [4] $! (3+3) ⇝ sumWithS   [4] 6
sumWithS      [] $! (6+4) ⇝ sumWithS   [] 10 ⇝ 10
```

Striktheit erzwingen

`$!` ist definiert durch das Primitiv `seq :: a -> b -> b`

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

`seq` erzwingt die Auswertung seines ersten Argumentes und liefert danach das zweite Argument zurück.

```
foo x =
  let zwischenwert = bar x
      ergebnis     = goo zwischenwert
  in  seq zwischenwert ergebnis
```

Von einigen Funktionen bietet die Standardbibliothek auch strikte Varianten, z.B. macht `foldl'` das gleiche wie `foldl`, erzwingt jedoch die Auswertung des Akkumulators in jedem Schritt.

Striktheit erzwingen

Das erste Argument von `seq` wird nur soweit ausgewertet, bis dessen äußere Form klar ist, darin kann auf weitere Thunks verwiesen werden:

Int, Bool: werden vollständig ausgewertet

Listen: ausgewertet bis klar ist, ob Liste leer ist oder nicht. Kopf und der Rumpf werden nicht ausgewertet!

Tupel: ausgewertet bis Tupel-Konstruktor fest steht, d.h. Elemente des Tupels werden nicht ausgewertet.

Maybe: ausgewertet bis `Nothing` oder `Just`, das Argument von Just wird noch nicht ausgewertet.

Generell wertet `seq` bis zum äußeren Konstruktor aus. Argumente des Konstruktors werden nicht weiter ausgewertet.

Schwache Kopf-Normalform, engl. Weak Head Normal Form

Demo

Demo `sumWith.hs`:
Wir führen `sumWithL` und `sumWithS` für große Listen mit GHC aus und werden überrascht!

- Wir haben gesehen, dass der Kompilier automatisch eine Striktheit-Analyse durchführt, d.h. wir müssen nur selten eingreifen.

- Wenn ein Stack-Overflow eintritt, dann sollte man über Endrekursion und auch über Striktheit nachdenken.

Zusammenfassung

- Terminations- und Ausnahmeverhalten können von der Auswertestrategie abhängen.
- Auswertungsstrategien call-by-name und call-by-value.
- Lazy evaluation als effiziente Implementierung von call-by-name.
- Lazy Evaluation erlaubt Verwendung unendlicher oder zirkulärer Datenstrukturen
- Lazy Evaluation erlaubt gute Modularisierung durch Trennung von Daten und Kontrollfluss
- Ein Funktionsargument, welches auf jeden Fall ausgewertet wird, heißt strikt.
- Annotationen zur Erzwingung von Striktheit.