2.6 Lazy Evaluation

In diesem Kapitel wollen wir die besondere Auswertungstrategie moderner funktionaler Sprachen genauer betrachten. Dazu benötigen wir zunächst einige Begriffe:

Redex (*re*ducible *ex*pression): Ein Ausdruck (Funktionsaufruf), auf den die linke Seite einer Gleichung passt (vgl. Kapitel 2.3).

Normalform: Ein Ausdruck ohne Redex (dies entspricht einem „Wert“).

Reduktionsschritt: Ersetze einen Redex durch eine entsprechend passende rechte Gleichungsseite

Das *Ziel einer funktionalen Auswertung* ist somit:

> Berechne eine Normalform durch Anwendung von Reduktionsschritten.

Das Problem hierbei ist: Ein Ausdruck kann viele Redexe enthalten! Welche Redexe soll man reduzieren?

Hier sind folgende Alternativen denkbar:

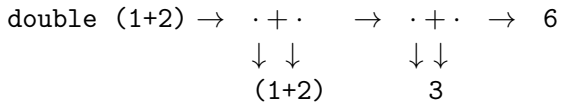
- „Sichere“ Strategie: Reduziere alle Redexe $\rightsquigarrow$ zu aufwändig
- Strikte Sprachen: Reduziere linken inneren Redex (*LI-Reduktion*)
- Nichtstrikte Sprachen: Reduziere linken äußeren Redex (*LO-Reduktion*)
  $\rightarrow$ Nachteil: Mehrfachauswertung von Argumenten

Beispiel:

```
double x = x+x

> double (1+2)  →  (1+2) + (1+2)  →  3 + (1+2)  →  3+3  →  6
```

Das Problem der Mehrfachauswertung kann durch eine *Graphrepräsentation* vermieden werden. Dabei werden bei mehrfachen Vorkommen von Variablen keine Terme dupliziert, sondern Variablen werden als Verweise auf einen Ausdruck interpretiert:



Die LO-Reduktion auf Graphen wird auch als **lazy evaluation** oder **call-by-need**-Reduktion bezeichnet. Diese hat die folgende Charakteristik:

1. Berechne einen Redex nur, falls er „benötigt“ wird
2. Berechne jeden Redex höchstens einmal

Wann wird nun aber ein Redex „benötigt“?

1. Durch ein Muster: sei die Funktion `f` durch folgende Regel definiert:

   ```
   f (x:xs) = ...
   ```

   Wenn wir nun den Aufruf

   ```
   f (g...)
   ```

   betrachten, dann wird der Wert (Konstruktor) von `(g...)` benötigt. Genauer wird dies ausgedrückt durch die Übersetzung von Mustern in case-Ausdrücke (vgl. Kapitel 2.3). Präziser:

   > Bei der Auswertung von “`case` t `of...`” wird der Wert von t benötigt.

2. Durch strikte Grundoperationen: Arithmetische Operationen und Vergleiche benötigen den Wert ihrer Argumente
   Dagegen: `if` e_1 `then` $e2$ `else` e_3 benötigt nur den Wert von e_1

3. Ausgabe: Drucken der benötigten Teile

Beim Fall 1 ist allerdings folgendes zu beachten:

> Der benötigte Redex wird nicht komplett ausgewertet, sondern nur bis zur **schwachen Kopfnormalform** (**SKNF**) (**weak head normal form**, **WHNF**).

Eine schwache Kopfnormalform ist hierbei eines der folgenden Möglichkeiten:

- $b \in$ `Int` $\cup$ `Float` $\cup$ `Bool` $\cup$ `Char` (primitiver Wert)
- $C\ e_1 \ldots e_k$, wobei C ein n-stelliger Konstruktor ist und $k \leq n$
- $(e_1, \ldots, e_k)$ (Tupel)
- $\backslash x \rightarrow e$ (lambda-Abstraktion)
- $f\ e_1 \ldots e_k$, wobei f eine n-stellige Funktion ist und $k < n$ (partielle Applikation)

Ein SKNF-Ausdruck ist also kein Redex, d.h. nicht weiter (an der Wurzel) reduzierbar.

Wichtig: Beim Pattern matching ($\simeq$ case-Ausdrücke) muss nur solange reduziert werden, bis eine schwache Kopfnormalform erreicht ist!
Bsp: Betrachte die Operation `take` n xs: berechne die ersten n Elemente von xs

```
take 0 xs     = []
take n (x:xs) = x : take (n-1) xs

take 1 ([1,2] ++[3,4])
```

benötigt

$\rightarrow \underbrace{\texttt{take 1 (1:([2] ++ [3,4]))}}_{\text{Redex}}$

$\rightarrow$ `1 : take` $\underbrace{\texttt{(1-1)}}_{\text{benötigt}}$ `([2] ++ [3,4])`

$\rightarrow$ `1 :` $\underbrace{\texttt{take 0 ([2] ++ [3,4])}}_{\text{Redex}}$

$\rightarrow$ `1 : []` (= `[1]`)

Beachte: das 2. Argument wurde nicht komplett ausgewertet (daher der Name „lazy“ evaluation).

Lazy evaluation hat Vorteile bei sog. *nicht-strikten* Funktionen (Vermeidung von überflüssigen Berechnungen):

Eine n-stellige Funktion f heißt **strikt** im i-ten Argument ($1 \leq i \leq n$), falls für alle $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$ gilt:

$$f\ x_1 \ldots x_{i-1}\ \bot\ x_{i-1} \ldots x_n = \bot$$

Hierbei bezeichnet $\bot$ eine nichtterminierende Berechnung (undefinierter Wert).

Z.B. ist

```
if-then-else :: Bool → a → a → a
```

strikt im 1. Argument, aber nicht strikt im 2. und 3. Argument

Vorteile von lazy evaluation:

- Vermeidung überflüssiger Auswertungen
  ($\leadsto$ *optimale Reduktionen*, [Huet/Lévy 79, Huet/Lévy 91])
- Rechnen mit unendlichen Datenstrukturen
  ($\leadsto$ Trennung von Daten und Kontrolle, $\leadsto$ Modularisierung, vgl. [Hughes 90])

Unendliche Datenstrukturen

Betrachten wir die Erzeugung einer (endlichen) Liste von ganzen Zahlen:

```
fromTo f t | f > t     = []
           | otherwise = f : fromTo (f + 1) t

fromTo 1 5  ⇝ [1, 2, 3, 4, 5]
```

Falls „`t` $= \infty$“ ist, dann ist das Ergebnis eine unendliche Liste, weil die Bedingung `f > t` nie wahr wird.

Spezialisierung von `fromTo` für `t` $= \infty$:

```
from f = f : from (f + 1)
```

```
from 1  ⇝ 1 : 2 : 3 : 4 : ... ^C
```

Obwohl `from 1` ein unendliches Objekt darstellt, kann man es wie andere Listen verwenden:

```
take 5 (from 1)  ⇝ [1, 2, 3, 4, 5]
```

Somit ist `fromTo` ein Spezialfall von `from`:

```
fromTo f t = take (t-f+1) (from f)
```

Vorteil: klare Trennung von Kontrolle (`take (t-f+1)`) und Datenerzeugung (`(from f)`)

Die Kontrolle und Datenerzeugung ist dagegen in der ersten Version von `fromTo` vermischt.

Häufig ist die unabhängige Datenerzeugung auch einfacher definierbar, wie wir unten an Beispielen zeigen werden.

Beachte: die Auswertung unendlicher Datenstrukturen wird erst durch lazy evaluation möglich! Betrachten wir dazu den Ausdruck

```
take 2 (from 1)
```

Bei einer strikten Auswertung (eager evaluation) wird versucht, das Argument `(from 1)` vollständig auszuwerten, was zu einer Endlosschleife führt. Dagegen geht eine lazy Berechnung wie folgt vor:

`take 2` $\underbrace{\texttt{(from 1)}}_{benötigt}$
→ `take 2` $\underbrace{\texttt{(1 : from (1+1))}}_{in\ SKNF}$
→ 1 : `take` $\underbrace{\texttt{(2-1)}}_{benötigt}$ `(from (1+1))`
→ 1 : `take 1` $\underbrace{\texttt{(from (1+1))}}_{benötigt}$
→ 1 : `take 1 ((1+1) : from ((1+1)+1))`
→ 1 : `(1+1) : take (1-1) (from ((1+1)+1))`
→ 1 : `(1+1) : take 0 (from ((1+1)+1))`
→ 1 : `(1+1) :` []
→ 1 : 2 : []

Beispiel: Erzeugung aller **Fibonacci-Zahlen**
Hierbei ist die n. Fibonacci-Zahl die Summe der beiden vorherigen

⇒ Im Gegensatz zu `from` benötigen wir hier zwei Parameter (die beiden vorherigen Fibonacci-Zahlen)

```
fibgen n1 n2 = n1 : fibgen n2 (n1 + n2)

fibs = fibgen 0 1
```

⇒ lineare (statt exponentielle) Laufzeit!

`fibs` $\rightsquigarrow$ `0 : 1 : 1: 2 : 3 : 5 : 8 ...`

Nun können wir die gewünschte Kontrolle hinzufügen, wie z.B.:

Berechne alle Fibonacci-Zahlen < 50:

```
takeWhile (<50) fibs  ⇝ [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Achtung: Filtern hier nicht möglich:

`filter (<10) fibs` $\rightsquigarrow$ `[0, 1, 2, 3, 5, 8,...` Endlosschleife!

Beispiel: Primzahlberechnung durch *Sieb des Eratosthemes*:
Idee:

1. Erstelle die Liste aller natürlicher Zahlen ≥ 2
2. Streiche darin alle Vielfachen bereits gefundener Primzahlen
3. Nächste kleinste Zahl ist prim

Beispiel:

$\underline{2}, \underline{3}, \not{4}, \underline{5}, \not{6}, \underline{7}, \not{8}, \not{9}, \not{10}, \underline{11}, \not{12}, \underline{13}, \ldots$

Wir realisieren dies mit Hilfe einer Funktion

```
sieve :: [Int]  → [Int]
```

Hierbei gelten die folgende Invarianten für `sieve`:

1. Das erste Element x der Eingabeliste ist prim und die restliche Eingabeliste enthält keine Vielfachen von Primzahlen $< x$.
2. Die Ergebnisliste besteht nur aus Primzahlen.

Mit diesen Überlegungen kann `sieve` wie folgt implementiert werden:

```
sieve :: [Int]  → [Int]
sieve (x:xs) = x : sieve (filter (\y  → y `mod` x > 0) xs)
```

Hierbei liefert das Prädikat in `filter` den Wert `False`, falls `y` ein Vielfaches von `x` ist. Insgesamt erhalten wir unter Benutzung von `sieve` die folgende Definition für die Liste aller Primzahlen:

```
primes :: [Int]
primes = sieve (from 2)
```

Die ersten 100 Primzahlen können wir dann wie folgt berechnen:

```
take 100 primes
```

Beispiel: **Hamming-Zahlen**:
Dies ist eine aufsteigende Folge von Zahlen der Form $2^a * 3^b * 5^c, a, b, c, \geq 0$ d.h. alle Zahlen mit Primfaktoren $2, 3, 5$

Idee:

1. 1 ist kleinste Hamming-Zahl.
2. Falls n Hamming-Zahl, dann ist auch $2n$, $3n$, $5n$ eine Hamming-Zahl.
3. Wähle aus den nächsten möglichen Hamming-Zahlen die kleinste aus und gehe nach 2.

Realisierung: Erzeuge die unendliche Liste der Hamming-Zahlen durch Multiplikation der Zahlen mit $2, 3, 5$ und mische die multiplizierten Listen aufsteigend geordnet.

Mischen zweier unendlicher(!) Listen in aufsteigender Folge:

```
ordMerge (x:xs) (y:ys)  |  x==y   =  x : ordMerge xs     ys
                        |  x < y  =  x : ordMerge xs     (y:ys)
                        |  x > y  =  y : ordMerge (x:xs) ys
```

Mischen dreier Listen `xs`, `ys`, `zs` durch "`ordMerge xs (ordMerge ys zs)`"
Somit erhalten wir die Gesamtlösung durch:

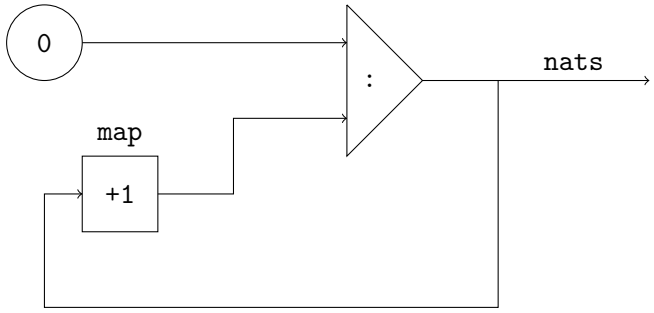
```
hamming = 1 : ordMerge (map (2*) hamming)
                       (ordMerge (map (*3) hamming)
                                 (map (*5) hamming))
```

Effizienz: `ordMerge` und `map` haben einen konstanten Aufwand für die Berechnung des nächsten Elements, sodass der Aufwand $0(n)$ für die Berechnung des n. Elements ist.

Berechne die ersten 15 Hammingzahlen durch:

```
take 15 hamming  ⇝ [1,2,3,4,5,6,8,9,10,12,15,16,18,20,24]
```

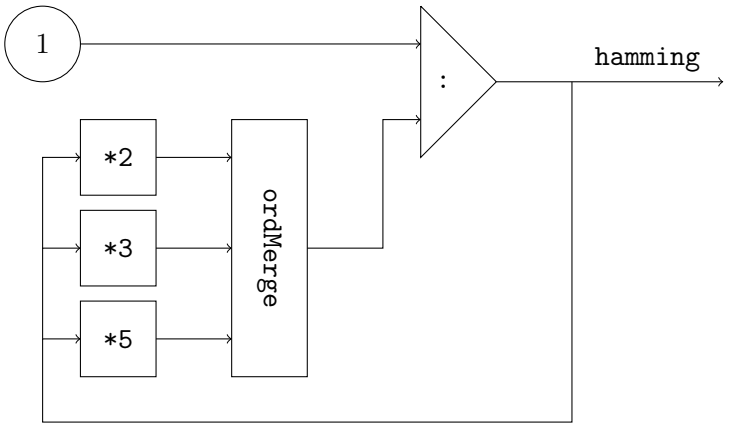
Da unendliche Listen viel Ähnlichkeit mit Strömen haben, kann man das Rechnen mit unendlichen Listen gut mittels *Prozessnetzen* planen. Z.B. entspricht dem Strom natürlicher Zahlen das folgende Netzwerk:



Hieraus ergibt sich die Haskell-Definition

```
nats = 0 : map (+1) nats
```

In ähnlicher Weise können wir das Netzwerk für Hamming-Zahlen beschreiben:



2.7 Deklarative Ein/Ausgabe

Wie kann man die Ein-/Ausgabe von Daten in einer deklarativen Sprache realisieren?

Eine Möglichkeit: Seiteneffekte wie in imperativen Sprachen (dies wird z.B. in Standard ML oder Scheme gemacht), z.B.

```
getInt :: Int
```

Seiteneffekt bei der Auswertung von `getInt`: Lesen einer Zahl von der Eingabe

Problematisch: Abhängigkeit von der Auswertungsreihenfolge, z.B.

```
inputDiff = getInt - getInt
```

Hier ist in der Regel `inputDiff` verschieden von 0!

Außerdem ist das Ergebnis abhängig von der Auswertungsreihenfolge von "`-`", z.B.

Eingabe: `5 3`
Ergebnis: entweder `2` oder `-2`!

Bei Lazy Evaluation ist die genaue Auswertungsreihenfolge schwer zu überschauen, daher ist hier ein I/O-Konzept ohne Seiteneffekte besonders wichtig!

Idee einer deklarativen Ein/Ausgabe: fasse I/O-Operationen als „Aktionen" auf, die die „äußere Welt" (vom Typ `World`) verändern/transformieren. Dies könnten wir durch folgenden Typ ausdrücken:

```
type IO a = World  →  (a,World)
```

Somit verändert eine *Aktion* vom Typ `IO a` den Weltzustand und liefert dabei ein Element vom Typ `a`.

Beispiele:

```
getChar :: IO Char         -- liest ein Zeichen
putChar :: Char  →  IO ()  -- schreibt ein Zeichen
```

Der Ergebnistyp `()` deutet an, dass `putChar` kein Ergebnis liefert, sondern nur den Weltzustand verändert.

Wichtig: Der Datentyp `World` bezeichnet den Zustand der gesamten äußeren Welt, er ist aber für den Programmierer nicht zugreifbar (insbesondere kann dieser nicht kopiert und auf zwei verschiedene Arten weiter transformiert werden!). Der Programmierer hat lediglich die Möglichkeit, Basisaktionen (wie z.B. `getChar`) zu größeren Einheiten zusammenzusetzen.

Komposition von Aktionen:
Da Aktionen Seiteneffekte auf der Welt sind, ist deren Ausführungsreihenfolge sehr wichtig. Aus diesem Grund ist es nur erlaubt, Aktionen sequenziell zusammenzusetzen. Hierzu gibt es einen Operator "`>>=`":

```
(>>=) :: IO a  →  (a  →  IO b)  →  IO b
```

„Definition" dieser Operation:

```
(a >>= fa) world = case a world of
                     (x,world')  →  fa x world'
```

Somit passiert durch die Abarbeitung von `(a >>= fa)` folgendes:

1. Führe Aktion `a` auf der aktuellen Welt aus.
2. Dies liefert ein Ergebnis `x` sowie eine veränderte Welt.

3. Führe die Aktion `(fa x)` auf der veränderten Welt aus.
4. Dies liefert ein Ergebnis `x'`, was auch das Gesamtergebnis ist.

Beispiel:

```
getChar >>= putChar
```

kopiert ein Zeichen von der Eingabe auf die Ausgabe.

Spezialfall: Komposition ohne Ergebnisverarbeitung:

```
(>>) :: IO a  → IO b  → IO b
a1 >> a2 = a1 >>= (\_  → a2)
```

„Leere Aktion“:

```
return :: a  → IO a -- mache nichts, liefere nur das Ergebnis
```

Beispiel: Ausgabe eines Strings (diese Aktion ist in Haskell so vordefiniert):

```
putStr :: String  → IO ()
putStr []     = return ()
putStr (c:cs) = putChar c >> putStr cs
```

Damit lautet das typische “Hello World”-Programm in Haskell:

```
main = putStr "Hello, World!"
```

Damit man I/O-Aktionen schöner aufschreiben kann, gibt es als „syntaktischen Zucker“ die `do`-**Notation**:

```
do p <- a1
   a2
```

steht für “`a1 >>= \p -> a2`”, und

```
do a1
   a2
```

steht für “`a1 >> a2`” (wobei nach dem “`do`” die Layout-Regel zur Anwendung kommt!).

Beispiel: Kopieren einer Eingabezeile:

```
echoLine = do c <- getChar
              putChar c
              if c=='\n' then return ()
                         else echoLine
```

Und nun noch einmal die Definition von `inputDiff`:

```
inputDiff = do i1 <- getInt
               i2 <- getInt
               putStr (show (i1-i2))
```

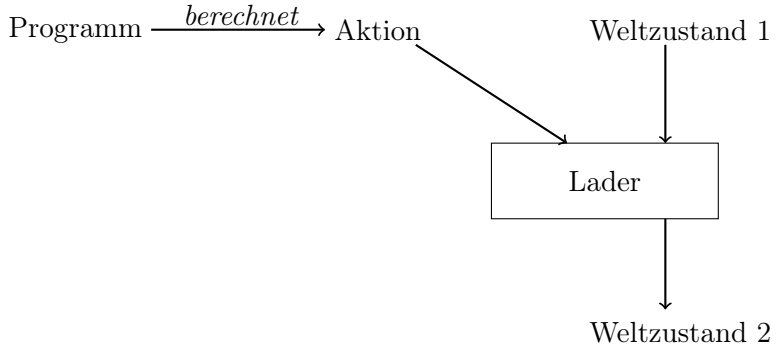
Hier ist die Reihenfolge der Berechnung explizit gemacht und damit nicht abhängig von der Auswertungsreihenfolge von "`-`".

Somit ist die Grundidee der deklarativen Ein/Ausgabe:

> Ein interaktives Programm berechnet eine Aktion(sfolge), diese verändert die Welt, *wenn diese auf einen Weltzustand angewendet wird.*

Aber wann passiert das?

Bei Ausführung des Programms durch das Betriebssystem! Daher wendet der Programmlader (Betriebssystem) die Aktion auf die momentane Welt an:



Vorteile dieses Konzepts:

- keine Seiteneffekte
- Aktionsreihenfolge ist explizit
- funktionale (referenziell transparente) Berechnung komplexer Aktionen möglich

Beispiel:

```
dupAct a = a >> a
```

Ausführung von `dupAct (putStr "Ha")` ergibt die Ausgabe "`HaHa`".
Dagegen in Standard ML:

```
fun dupAct a = (a ; a)
```

Ausdruck `dupAct (print "Ha")` ergibt die Ausgabe "`Ha`".

Somit können durch die deklarative Ein/Ausgabe die üblichen funktionalen Programmiertechniken auch zum Rechnen mit Aktionen benutzt werden.

2.8 Zusammenfassung

Wir fassen noch einmal die Vorteile funktionaler Sprachen zusammen:

- einfaches Modell: Ersetzung von Ausdrücken
- überschaubare Programmstruktur:
  Funktionen, definiert durch Gleichungen für Spezialfälle (pattern matching)
- Ausdrucksmächtigkeit: Programmschemata mittels Funktionen höherer Ordnung ausdrückbar
- Sicherheit: keine Laufzeittypfehler
- Wiederverwendung: Polymorphismus + Funktion höherer Ordnung
- Modularität und Optimalität durch Lazy Evaluation
- Verifikation und Wartbarkeit:
  keine Seiteneffekte: erleichtert Verifikation und Transformation