Funktionales Programmieren
(Praktische Informatik 3)

Berthold Hoffmann
Studiengang Informatik
Universität Bremen

Winter 03/04

Vorlesung vom 08.12.2003: Verzögerte Auswertung und unendliche Datenstrukturen

Inhalt

- Verzögerte Auswertung
  - . . . und was wir davon haben.
- Unendliche Datenstrukturen
  - . . . und wozu sie nützlich sind.
- Fallbeispiel: Parserkombinatoren

Verzögerte Auswertung

- Auswertung: Reduktion von Gleichungen
  - Strategie: Von außen nach innen; von links nach rechts
- Effekt: Parameterübergabe *call by need*, nicht-strikt
  - Beispiel:

    ```
    silly x y = y; double x = x+ x
    ```

    ```
    silly (double 3) (double 4)
    ```
    $\rightsquigarrow$ `double 4` $\rightsquigarrow$ `4+ 4` $\rightsquigarrow$ 8
  - Das erste Argument von `silly` wird gar nicht ausgewertet.
  - Das zweite Argument von `silly` wird erst im Funktionsrumpf ausgewertet.

Striktheit

- Eine Funktion `f` ist strikt in einem Argument `x`, wenn ein undefinierter Wert für `x` die Funktion immer undefiniert werden läßt.
- Beispiele:
  - `(+)` ist strikt in beiden Argumenten.
  - `(&&)` ist strikt im ersten, nicht-strikt im zweiten.
    `False && (1/0 == 0)` $\rightsquigarrow$ `False`
  - `silly` ist nicht-strikt im ersten Argument. `silly (1/0) 3` $\rightsquigarrow$ `3`

Effekte der verzögerten Auswertung

- Datenorientierte Programme:
  - Berechung von $\sum_{i=1}^{n} \frac{1}{i}$
    - ▷ Bilde Liste der Zahlen von 1 bis n (`fromTo 1 n`)
    - ▷ Bilde Kehrwert jedess Listenelement (`map (1/)`)
    - ▷ Bilde Summe

```
sum []     = 0
sum (h:t) = h + sum t
map f [] = []
map f (h:t) = f h : map f t
fromTo m n | m > n      = []
           | otherwise = m : fromTo (succ m) n

reciprocalSum :: (Enum a,Ord a,Fractional a) => a -> a
reciprocalSum n = sum (map (1/) (fromTo 1 n))
```

$\Longrightarrow$ Es werden keine Listen von Zwischenergebnissen erzeugt.

```
sum (map (1/) (fromTo 1 n))
```

$\Longrightarrow$ Es werden keine Listen von Zwischenergebnissen erzeugt.

```
    sum (map (1/) (fromTo 1 n))
⇝   sum (map (1/) (1 : fromTo (succ 1) n))
```

$\Longrightarrow$ Es werden keine Listen von Zwischenergebnissen erzeugt.

```
    sum (map (1/) (fromTo 1 n))
~>  sum (map (1/) (1 : fromTo (succ 1) n))
~>  sum (1 : map (1/) (fromTo (succ 1) n))
```

$\Longrightarrow$ Es werden keine Listen von Zwischenergebnissen erzeugt.

```
    sum (map (1/) (fromTo 1 n))
~>  sum (map (1/) (1 : fromTo (succ 1) n))
~>  sum (1 : map (1/) (fromTo (succ 1) n))
~>  1 + sum (map (1/) (fromTo (succ 1) n))
```

$\Longrightarrow$ Es werden keine Listen von Zwischenergebnissen erzeugt.

```
    sum (map (1/) (fromTo 1 n))
⇝   sum (map (1/) (1 : fromTo (succ 1) n))
⇝   sum (1 : map (1/) (fromTo (succ 1) n))
⇝   1 + sum (map (1/) (fromTo (succ 1) n))
⇝   1 + sum (map (1/) (fromTo 2 n))
```

$\Longrightarrow$ Es werden keine Listen von Zwischenergebnissen erzeugt.

```
    sum (map (1/) (fromTo 1 n))
⇝   sum (map (1/) (1 : fromTo (succ 1) n))
⇝   sum (1 : map (1/) (fromTo (succ 1) n))
⇝   1 + sum (map (1/) (fromTo (succ 1) n))
⇝   1 + sum (map (1/) (fromTo 2 n))
⇝   1 + sum (map (1/) (2 : fromTo (succ 2) n))
```

$\Longrightarrow$ Es werden keine Listen von Zwischenergebnissen erzeugt.

```
    sum (map (1/) (fromTo 1 n))
⇝   sum (map (1/) (1 : fromTo (succ 1) n))
⇝   sum (1 : map (1/) (fromTo (succ 1) n))
⇝   1 + sum (map (1/) (fromTo (succ 1) n))
⇝   1 + sum (map (1/) (fromTo 2 n))
⇝   1 + sum (map (1/) (2 : fromTo (succ 2) n))
⇝   1 + sum(0.5 : map (1/) (fromTo (succ 2) n))
```

$\Longrightarrow$ Es werden keine Listen von Zwischenergebnissen erzeugt.

```
    sum (map (1/) (fromTo 1 n))
⇝   sum (map (1/) (1 : fromTo (succ 1) n))
⇝   sum (1 : map (1/) (fromTo (succ 1) n))
⇝   1 + sum (map (1/) (fromTo (succ 1) n))
⇝   1 + sum (map (1/) (fromTo 2 n))
⇝   1 + sum (map (1/) (2 : fromTo (succ 2) n))
⇝   1 + sum(0.5 : map (1/) (fromTo (succ 2) n))
⇝   1 + 0.5 + sum (map (1/) (fromTo (succ 2) n))
```

$\Longrightarrow$ Es werden keine Listen von Zwischenergebnissen erzeugt.

```
    sum (map (1/) (fromTo 1 n))
~>  sum (map (1/) (1 : fromTo (succ 1) n))
~>  sum (1 : map (1/) (fromTo (succ 1) n))
~>  1 + sum (map (1/) (fromTo (succ 1) n))
~>  1 + sum (map (1/) (fromTo 2 n))
~>  1 + sum (map (1/) (2 : fromTo (succ 2) n))
~>  1 + sum(0.5 : map (1/) (fromTo (succ 2) n))
~>  1 + 0.5 + sum (map (1/) (fromTo (succ 2) n))
~>  1.5 + sum (map (1/) (fromTo (succ 2) n))
```

- Knapper kan man `reciprocalSum` so definieren:

```
reciprocalSum n = sum [ 1/i | i<- [1..n]]
```

Unendliche Datenstrukturen

Unendliche Datenstrukturen: Ströme

- Ströme sind unendliche Listen.
  - Unendliche Liste `[2,2,2,...]`

    ```
    twos = 2 : twos
    ```

  - Die natürlichen Zahlen:

    ```
    nat  = [1..]
    ```

  - Bildung von unendlichen Listen:

    ```
    cycle :: [a]-> [a]
    cycle xs = xs ++ cycle xs
    ```

- Repräsentation durch endliche, zyklische Datenstruktur
  - Kopf wird nur einmal ausgewertet. Zeigen

    ```
    head(tail(tail(cycle (trace "1 x CONS: " "x"))))
    ```

Bsp: Berechnung der ersten n Primzahlen

- Eratosthenes — bis wo sieben?
- Lösung: Berechnung aller Primzahlen,

```
sieve :: [Integer]-> [Integer]
-- sieve []   = []
sieve (p:ps) =
  p:(sieve (filter (\n-> n `mod` p /= 0) ps))
```

- Keine Rekursionsverankerung (Testen)

```
primes :: Int-> [Integer]
primes n = take n (sieve [2..])
```

Bsp: Fibonacci-Zahlen

- Aus der Kaninchenzucht.
- Sollte jeder Informatiker kennen.

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

- Problem: exponentieller Aufwand.

- Lösung: zuvor berechnete Teilergebnisse wiederverwenden.
- Sei `fibs :: [Integer]` Strom aller Fib'zahlen:

```
                fibs  1  1  2  3  5  8 13 21 34 55
           tail fibs     1  2  3  5  8 13 21 34 55
tail (tail fibs)        2  3  5  8 13 21 34 55
```

- Damit ergibt sich:

- Lösung: zuvor berechnete Teilergebnisse wiederverwenden.
- Sei `fibs :: [Integer]` Strom aller Fib'zahlen:

```
            fibs  1  1  2  3  5  8 13 21 34 55
       tail fibs     1  2  3  5  8 13 21 34 55
tail (tail fibs)        2  3  5  8 13 21 34 55
```

- Damit ergibt sich:

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

- Aufwand: linear, da `fibs` nur einmal ausgewertet wird.
- `n`-te Fibonnacizahl mit `fibs !! n` Zeigen

Tücken der verzögerten Auswertung

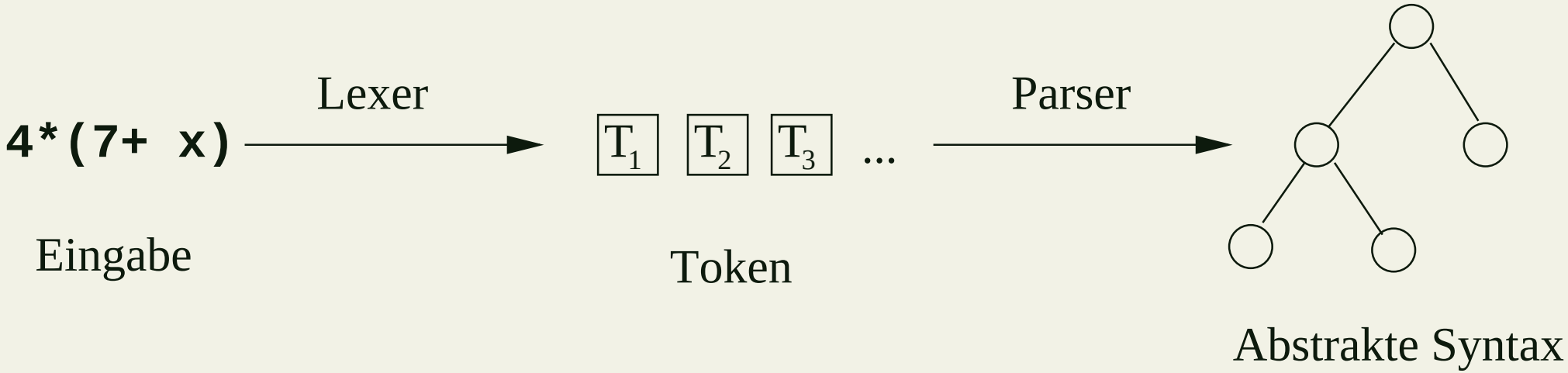
- Auch Fehlermeldungen werden verzögert
- Beispiel: Warenlager (Übungsblatt 2)
  - `remove karstadt ("Glueck",1)` oder
    `remove karstadt ("CD",-3)` führen zu Fehlern
  - `lookup (remove karstadt ("Glueck" 1)),"Hose"` funktioniert trotzdem
  - Knallen tut's erst bei `prnt (remove karstadt ("CD",-3))`

Zeigen

Fallstudie: Parsierung mit Parserkombinatoren

Parsieren im Allgemeinen

- Gegeben: Grammatik
- Gesucht: Funktion, die Wörter der Grammatik erkennt



- Ein Parser bildet Eingabe auf Syntaxbäume ab.
  - Mehrere Syntaxbäume sind möglich.
  - Backtracking ist möglich.
  - Durch verzögerte Auswertung bleibt es dennoch effizient.
- Lexer erkennen Terminalsymbole
- Kombinatoren für Parser:
  - Sequenzen (erst A, dann B)
  - Alternativen (A oder B)
  - Kombinatoren (z.B. Listen A^*, nicht-leere Listen A^+)

Arithmetische Ausdrücke

- Grammatik:

```
Expr   ::= Term '+' Expr
         | Term '-' Expr
         | Term
Term   ::= Factor '*' Term
         | Factor '/' Term
         | Factor
Factor ::= Number | '(' Expr ')'
Number ::= Digit  | Digit Number
Digit  ::= '0 '|...| '9'
```

- Daraus abstrakte Syntax:

```
data Expr   = Plus  Expr Expr
            | Minus Expr Expr
            | Times Expr Expr
            | Div   Expr Expr
            | Number Int
            deriving (Eq, Show)
```

  - Unterscheidung von `Expr`, `Term`, `Factor`, ist hier unnötig.

Modellierung in Haskell

- Welchen Typ haben Parser?
  - Parser übersetzen eine Tokenliste in abstrakte Syntaxbäume
  - Sie sind parametrisiert mit Eingabetyp `a` und Ergebnistyp `b`

Modellierung in Haskell

- Welchen Typ haben Parser?
  - Parser übersetzen eine Tokenliste in abstrakte Syntaxbäume
  - Sie sind parametrisiert mit Eingabetyp `a` und Ergebnistyp `b`
  - Müssen mehrdeutige Ergebnisse modellieren können

Modellierung in Haskell

- Welchen Typ haben Parser?
  - Parser übersetzen eine Tokenliste in abstrakte Syntaxbäume
  - Sie sind parametrisiert mit Eingabetyp `a` und Ergebnistyp `b`
  - Müssen mehrdeutige Ergebnisse modellieren können
  - Müssen Rest der Eingabe modellieren können

Modellierung in Haskell

- Welchen Typ haben Parser?
  - Parser übersetzen eine Tokenliste in abstrakte Syntaxbäume
  - Sie sind parametrisiert mit Eingabetyp `a` und Ergebnistyp `b`
  - Müssen mehrdeutige Ergebnisse modellieren können
  - Müssen Rest der Eingabe modellieren können

```
type Parse a b = [a]-> [(b, [a])]
```

- Beispiel:

```
parse "3+4*5" ⇝ [ (3, "+4*5"),
                  (Plus 3 4, "*5"),
                  (Plus 3 (Times 4 5), "")]
```

Basisparser

- Erkennt nichts:

```
none :: Parse a b
none = const []
```

- Erkennt alles:

```
suceed :: b-> Parse a b
suceed b inp = [(b, inp)]
```

- Erkennt einzelne Zeichen:

```
spot  :: (a-> Bool)-> Parse a a
spot p []     = []
spot p (x:xs) = if p x then [(x, xs)] else []

token :: Eq a=> a-> Parse a a
token t = spot (== t)
```

Kombinatoren

- Alternative:

```
infixl 3 <|>
(<|>) :: Parse a b-> Parse a b-> Parse a b
(p1 <|> p2) i = p1 i ++ p2 i
```

- Sequenz:
  - Rest des ersten Parsers als Eingabe für den zweiten

```
infixl 5 >*>
(>*>) :: Parse a b-> Parse a c-> Parse a (b, c)
(p1 >*> p2) i = [((y, z), r2) | (y, r1) <- p1 i,
                                (z, r2) <- p2 r1]
```

- Eingabe weiterverarbeiten:

```
infix 4 <*>
(<*>) :: Parse a b-> (b-> c)-> Parse a c
(p <*> f) inp = [(f x, r) | (x, r)<- p inp]
```

- Damit z.B. Sequenzierung rechts/links:

```
infixl 5 *>, >*
(*>) :: Parse a b-> Parse a c-> Parse a c
(>*) :: Parse a b-> Parse a c-> Parse a b
p1 *> p2 = p1 >*> p2 <*> snd
p1 >* p2 = p1 >*> p2 <*> fst
```

Abgeleitete Kombinatoren

- Listen: $A^* ::= AA^* \mid \varepsilon$

```
list :: Parse a b-> Parse a [b]
list p = p >*> list p <*> uncurry (:)
         <|> suceed []
```

- Nicht-leere Listen: $A^+ ::= AA^*$

```
some :: Parse a b-> Parse a [b]
some p = (p >*> list p) <*> uncurry (:)
```

- Erinnerung: `uncurry (:) (a,b) = a : b`

Einschub: Präzedenzen

Höchste Priorität: Funktionsapplikation

```
infixr 9  .
infixr 8  ^, ^^, **
infixl 7  *, /, `quot`, `rem`, `div`, `mod`
infixl 6  +, -
infixr 5  : ++                                  >*> >* *>
infix  4  ==, /=, <, <=, >=, >                  <*>
infixr 3  &&                                    <|>
infixr 2  ||
infixl 1  >>, >>=
```

Der Kern des Parsers

- Parse von `Expr`

```
pExpr :: Parse Char Expr
pExpr = pTerm >* token '+' >*> pExpr
                              <*> uncurry Plus
        <|> pTerm >* token '-' >*> pExpr
                              <*> uncurry Minus
        <|> pTerm
```

- Parse von `Term`

```
pTerm :: Parse Char Expr
pTerm = pFactor >* token '*' >*> pTerm
                              <*> uncurry Times
        <|> pFactor >* token '/' >*> pTerm
                                 <*> uncurry Div
        <|> pFactor
```

- Parse von `Factor`

```
pFactor :: Parse Char Expr
pFactor =
        some (spot isDigit) <*> Number. read
        <|> token '(' *> pExpr >* token ')'
```

Die Hauptfunktion

- Lexing: Leerzeichen aus der Eingabe entfernen
- Zu prüfen: Testen
  - Parser liest die Eingabe vollständig
  - der Parse ist eindeutig

```
parse :: String-> Expr
parse i = case filter (null. snd)
                      (pExpr[c| c<-i, not(isSpace c)])
              of []       -> error "Wrong input."
                 [(e,_)] -> e
                 _        -> error "Ambiguous input."
```

Zusammenfassung Parserkombinatoren

- Systematische Konstruktion des Parsers aus der Grammatik.
- Abstraktion durch Funktionen höherer Ordnung.
- Durch verzögerte Auswertung annehmbare Effizienz:
  - Einfache Implementierung (wie oben) skaliert nicht
  - Grammatik darf nicht linksrekursiv sein ($\Longrightarrow$ terminiert nicht!)
  - Grammatik muß eindeutig sein (LL(1) o.ä.)
  - Gut implementierte Bibliotheken (wie Parsec) sind bei eindeutiger Grammatik auch für große Eingaben geeignet

- Wir brauchen noch mehr Möglichkeiten zum Aufbau der Syntaxbäume!

Zusammenfassung

- Verzögerte Auswertung erlaubt unendliche Datenstrukturen
  - Zum Beispiel: Ströme (unendliche Listen)
- Parserkombinatoren:
  - Systematische Konstruktion von Parsern
  - Durch verzögerte Auswertung annehmbare Effizienz