Vorlesung vom 13.12.2004: 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: *call-by-need* Parameterübergabe, nicht-strikt
  - Beispiel:

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

    `silly (double 3) (double 4)` $\rightsquigarrow$ `double 4` $\rightsquigarrow$ `4+ 4` $\rightsquigarrow$ `8`
  - Erstes Argument von `silly` wird nicht ausgewertet.
  - Zweites Argument von `silly` wird erst im Funktionsrumpf ausgewertet.

Striktheit

- Funktion `f` ist strikt in einem Argument `x` gdw.

$$\texttt{x} \equiv \bot \Longrightarrow \texttt{f(x)} \equiv \bot$$

- Beispiele:
  - `(+)` strikt in beiden Argumenten.
  - `(&&)` strikt im ersten, nicht-strikt im zweiten.
    `False && error` $\rightsquigarrow$ `False`
  - `silly` nicht-strikt im ersten. `silly error 3` $\rightsquigarrow$ 3
  - Zeigen.
- Haskell ist
  - als nicht-strikt spezifiziert;
  - (meist) mit verzögerter Auswertung implementiert.

Effekte der Nichtstriktheit

Datenorientierte Programme

- Berechnung der Summe von `1` bis `n`:

```
sqrsum n = sum (map (^2) [1..n])
```

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

- Minimum einer Liste durch

```
min' xs = head (msort xs)
```

  $\Longrightarrow$ Liste wird nicht vollständig sortiert — binäre Suche.

- Rucksackproblem: Elemente auswählen, so dass Wert maximiert:

```
select :: [(Double, Double)]-> Double-> [Int]
select elts w =
  let wgh i = fst (elts !! (i-1))
      val i = snd (elts !! (i-1))
      permissible seq  = sum (map wgh seq) <= w
      better seq1 seq2 = compare (sum (map val seq2))
                                 (sum (map val seq1))
  in head (sortBy better
             (filter permissible (seqs (length elts))))

seqs :: Int-> [[Int]]
seqs 0 = [[]]
seqs n = seqs (n-1) ++ map (n :) (seqs (n-1))
```

Unendliche Datenstrukturen: Ströme

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

    ```
    twos = 2 : twos
    ```

  - Liste der 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.

    ```
    cycle (trace "Foo!" "x")
    ```

- Nützlich für Listen mit unbekannter Länge

Bsp: Berechnung der ersten n Primzahlen

- Eratosthenes — aber bis wo sieben?
- Lösung: Berechnung aller Primzahlen, davon die ersten n.

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

- Keine Rekursionsverankerung (vgl. alte Version)

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

Testen.

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
```

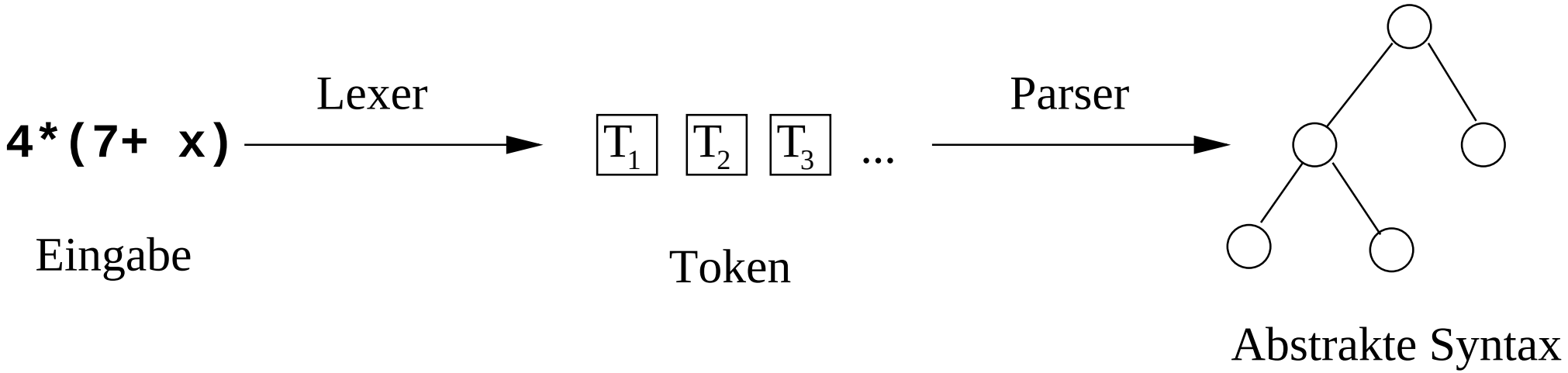
- Damit ergibt sich:

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

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

Fallstudie: Parsierung

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



Parser

- Parser bilden Eingabe auf Parsierungen ab
  - Mehrere Parsierungen möglich
  - Backtracking möglich
  - Durch verzögerte Auswertung dennoch effizient
- Basisparser erkennen Terminalsymbole
- Parserkombinatoren erkennen Nichtterminalsymbole
  - Sequenzierung (erst A, dann B)
  - Alternierung (entweder A oder B)
  - Abgeleitete Kombinatoren (z.B. Listen A^*, nicht-leere Listen A^+)

Grammatik für Arithmetische Ausdrücke

$$
\begin{aligned}
\mathit{Expr} &::= \mathit{Term}\ \texttt{+}\ \mathit{Term} \mid \mathit{Term}\ \texttt{-}\ \mathit{Term} \mid \mathit{Term} \\
\mathit{Term} &::= \mathit{Factor}\ \texttt{*}\ \mathit{Factor} \mid \mathit{Factor}\ \texttt{/}\ \mathit{Factor} \mid \mathit{Factor} \\
\mathit{Factor} &::= \mathit{Number} \mid \mathit{Variable} \mid (\mathit{Expr}) \\
\mathit{Number} &::= \mathit{Digit}^{+} \\
\mathit{Digit} &::= \texttt{0} \mid \cdots \mid \texttt{9} \\
\mathit{Var} &::= \mathit{Char}^{+} \\
\mathit{Char} &::= \texttt{a} \mid \cdots \mid \texttt{z} \mid \texttt{A} \mid \cdots \mid \texttt{Z}
\end{aligned}
$$

Abstrakte Syntax für Arithmetische Ausdrücke

- Zur Grammatik abstrakte Syntax (siehe altes Beispiel)

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

- Hier Unterscheidung `Term`, `Factor`, `Number` unnötig.

Modellierung in Haskell

Welcher Typ für Parser?

- Parser übersetzt Token in abstrakte Syntax
- Parametrisiert über Eingabetyp (Token) `a` und Ergebnis `b`
- Müssen mehrdeutige Ergebnisse modellieren
- Müssen Rest der Eingabe modellieren

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

Basisparser

- Erkennt nichts:

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

- Erkennt alles:

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

- Erkennt einzelne Zeichen:

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

Basiskombinatoren

- Alternierung:

```
infixl 3 `alt`
alt :: Parse a b-> Parse a b-> Parse a b
alt p1 p2 i = p1 i ++ p2 i
```

- Sequenzierung:
  - 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]
```

Basiskombinatoren

- Eingabe weiterverarbeiten:

```
infix 4 `use`
use :: Parse a b-> (b-> c)-> Parse a c
use 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 `use` snd
p1 >* p2 = p1 >*> p2 `use` fst
```

Abgeleitete Kombinatoren

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

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

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

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

- NB. Präzedenzen: `>*>` (5) vor `use` (4) vor `alt` (3)

Einschub: Präzedenzen der Operatoren in Haskell

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  >>, >>=
infixr 1  =<<
infixr 0  $, $!, `seq`
```

Parsierung Arithmetische Ausdrücke

- Token: `Char`
- Parsierung von `Expr`

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

- Parsierung von `Term`

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

Der Kern des Parsers

- Parsierung von `Factor`

```
pFactor :: Parse Char Expr
pFactor = some (spot isDigit) `use` Number. read
    `alt` some (spot isAlpha) `use` Var
    `alt` token '(' *> pExpr >* token ')'
```

Die Hauptfunktion

- Lexing: Leerzeichen aus der Eingabe entfernen
- Zu prüfen:
  - Parsierung konsumiert Eingabe
  - Keine Mehrdeutigkeit Testen.

```
parse :: String-> Expr
parse i =
  case filter (null. snd)
         (pExpr (filter (not.isSpace) i)) of
    [] -> error "Input does not parse."
    [(e, _)] -> e
    _  -> error "Input is ambiguous."
```

Ein kleiner Fehler

- Mangel: 3+4+5 ist Syntaxfehler — Fehler in der Grammatik
- Behebung: leichte Änderung der Grammatik . . .

$$
\begin{array}{rcl}
\mathit{Expr} & ::= & \mathit{Term}\ \texttt{+}\ \mathit{Expr} \mid \mathit{Term}\ \texttt{-}\ \mathit{Expr} \mid \mathit{Term} \\
\mathit{Term} & ::= & \mathit{Factor}\ \texttt{*}\ \mathit{Term} \mid \mathit{Factor}\ \texttt{/}\ \mathit{Term} \mid \mathit{Factor} \\
\mathit{Factor} & ::= & \mathit{Number} \mid \mathit{Variable} \mid (\mathit{Expr}) \\
\mathit{Number} & ::= & \mathit{Digit}^{+} \\
\mathit{Digit} & ::= & \texttt{0} \mid \cdots \mid \texttt{9} \\
\mathit{Var} & ::= & \mathit{Char}^{+} \\
\mathit{Char} & ::= & \texttt{a} \mid \cdots \mid \texttt{z} \mid \texttt{A} \mid \cdots \mid \texttt{Z}
\end{array}
$$

  - (vergleiche alt)
- Abstrakte Syntax bleibt . . .

- Entsprechende Änderung des Parsers in `pExpr`

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

- `...` und in `pTerm`:

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

  - (vergleiche alt)

- `pFactor` und Hauptfunktion bleiben: Testen.

Zusammenfassung Parserkombinatoren

- Systematische Konstruktion des Parsers aus der Grammatik.
- Abstraktion durch Funktionen höherer Ordnung.
- Durch verzögerte Auswertung annehmbare Effizienz:
  - Grammatik muß eindeutig sein (LL(1) o.ä.)
  - Vorsicht bei Mehrdeutigkeiten!
  - Effizient implementierte Büchereien mit gleicher Schnittstelle auch für große Eingaben geeignet.

Zusammenfassung

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