Funktionale Programmierung und Typtheorie

5. Fortgeschrittene Konzepte

5. Fortgeschrittene Konzepte

5.6 Parser

Grundlegender Typ eines Parsers
⇒ `type Parser = String -> Tree`
⇒ `parse :: Parser`

Weitere Modifikation:
Ein Parser muß nicht unbedingt den kompletten Eingabestring verbrauchen. Das Resultat muß auch den nicht genutzten Teil der Eingabe beinhalten.

⇒ `type Parser = String -> (Tree, String)`

5.6 Parser

Die Regeln einer Grammatik können mehrdeutig sein. Das heißt, dass es unter Umständen mehrere Wege gibt, den zu analysierenden Text zu erzeugen (und damit auch mehrere Erzeugungsbäume, die aus dem Text rekonstruiert werden können).
⇒ `type Parser = String -> [(Tree, String)]`

Ein Parser liefert auch andere Objekte als Bäume zurück.
`type Parser a = String -> [(a, String)]`

A parser for things
Is a function from strings
To lists of pairs
Of things and strings.

5.6 Parser

Wir haben in der letzten Vorlesung so genannte *elementare Parser* kennengelernt.

Parser *item*, der ein beliebiges Zeichen akzeptiert und dieses als Parsingergebnis zurückliefert.

```
item :: Parser Char
item [ ]     = [ ]
item (x: xs) = [(x,xs)]
```

Beispiel

```
item "sturm" ⇝ [('s',"turm")]
item " sturm" ⇝ [(' ',"sturm")]
item " " ⇝ [ ]
```

5.6 Parser

Es gibt nun weitere Parser, die man als grundlegend bezeichnen kann, z.B. einen Parser *returnP*, der *jede Eingabe s* akzeptiert und immer ein Ergebnis *e* liefert ohne ein Zeichen zu verbrauchen.

```
returnP :: a -> Parser a
returnP e = λs -> [(e,s)]
```

Beispiel

```
returnP 2009 "Weihnachten" ⇝ [(2009,"Weihnachten")]
```


5.6 Parser

Der Parser *symbol* überprüft, ob das erste Zeichen der Eingabe mit dem Argument c übereinstimmt. Wenn ja, so ergibt sich ein Paar aus dem erkannten x und dem nicht bearbeitetem Reststring xs.

Achtung `type parser tok a = [tok] -> [(a,[tok])]`

```
symbol :: Eq a => a -> Parser a a
symbol c [ ]     = [ ]
symbol c (x: xs) = [(x, xs)| c == x]
```

Der Parser *token*, der ein ganzes Wort erkennen kann, bekommt dieses als Argument übergeben.

```
token :: Eq a => [a] -> Parser a [a]
token k xs | k == take n xs = [(k, drop n xs)]
           | otherwise      = [ ]
          where n = length k
```


5.6 Parser

- Mittels Kombination können aus *elementaren Parsern* weitere Parser erzeugt werden. Man spricht von *Parser Komposition*.
- Dazu sind Operationen höherer Ordnung erforderlich, die bei Eingabe von einem oder mehreren Parsern einen neuen Parser als Ergebnis liefern.

Sequentielle Verkettung von Parsern

$L_1 L_2 = \{w_1 w_2 \mid w_1 \in L_1,\ w_2 \in L_2\}$

```
(+.+) :: Parser tok a -> Parser tok b -> Parser tok (a,b)

(p1 +.+ p2) toks =
[((v1,v2),rest2) | (v1,rest1) <- p1 toks, (v2,rest2) <- p2 rest1]
```


5.6 Parser

Alternative oder Vereinigung von Sprachen

Vereinigung der Sprachen L_1 und L_2 zu $L_1 \cup L_2$

```
(|||) :: Parser tok a -> Parser tok a -> Parser tok a

(p1 ||| p2) toks = p1 toks ++ p2 toks
```

Anmerkung

||| ist nicht kommutativ. Vereinigung ist damit nicht vollständig.

Manipulation von Parsern

```
(<<<) :: Parser tok a -> (a -> b) -> Parser tok b
(p <<< f) toks = [(f v,rest) | (v,rest) <- p toks]
```

Anmerkung

Ergebnisse werden mittels f transformiert.

5.6 Parser

Beispiele

- `> (token "ab" (+.+) symbol 'a') "abab"`
- `> (token "ab" ||| token "aba") "abab"`

5.6 Parser

Beispiele

- `> (token "ab" (+.+) symbol 'a') "abab"`
  `[(("ab",'a'),"b")]`
- `> (token "ab" ||| token "aba") "abab"`
  `[("ab","ab"),("aba","b")]`

5.6 Parser

Beispiele

- `sp :: Parser Char a ->  Parser Char a`
  `sp p = p . dropWhile (== '')`
- `just :: Parser s a -> Parser s a`
  `just p = filter (null . snd) . p`

5.6 Parser

Beispiele

- `sp :: Parser Char a ->  Parser Char a`
  `sp p = p . dropWhile (== '')`
  Funktion *sp* modifiziert einen Parser derart, dass er führende Leerzeichen erlaubt.
- `just :: Parser s a -> Parser s a`
  `just p = filter (null.fst) . p`
  Die Funktion *just* modifiziert einen Parser derart, dass er nur noch Eingaben akzeptiert, die er komplett verarbeiten kann. Der Reststring muß bei allen Eingaben leer sein.

5.7 Konstruktorklassen

Typvariablen können nicht nur über Typen variieren, sondern auch über Typkonstruktoren.

Definition (Konstruktorausdrücke)

Konstruktorausdrücke sind entweder Konstruktorvariablen, Typkonstruktoren oder Konstruktorapplikationen.

$K \ ::= \ \alpha \ | \ C \ | \ (K_1 \ K_2)$

Ein *Kind* ist für einen Konstruktor so etwas Ähnliches wie ein Typ für einen Datenwert.

Jeder gewöhnliche Typ hat Kind `*`. Der Listenkonstruktor [ ] bildet Typen auf Typen ab und hat daher einen Kind `* -> *`.

5.7 Konstruktorklassen

Beispiel (Typkonstruktoren)

`* -> *` Typkonstruktor mit einem Argument
`* -> * -> *` Typkonstruktor mit zwei Argumenten

- Kinds spielen bei der Konstruktorapplikation eine wesentliche Rolle.
- Hat K_1 das Kind $k_2 \to k_1$ und K_2 das Kind k_2, so hat die Applikation $(K_1 K_2)$ Kind k_1.
- Im anderen Fall wäre $(K_1 K_2)$ nicht wohlgeformt und würde vom System zurückgewiesen werden.

Wozu benötigt man in der Programmierung die Betrachtung von Kinds und damit die Betrachtung von Typen für Konstruktoren?

5.7 Konstruktorklassen

Beispiel (Programmierung eines generischen Datentyps Baum)

Generische Datentyp Baum kann zu binären, ternären und beliebigen Bäumen gemacht werden.

```
data XTree sons a = XMt  | XBranch a (sons (Xtree sons a))
```

```
XTree  hat Kind (* -> *) -> * -> *
sons   hat Kind (* -> *)
a      hat Kind  *
```

`XTree Pair` stellt den Typ der Binärbäume , `XTree []` den Typ der allgemeinen Bäume und `XTree TwoThree` den Typ der 2-3-Bäume dar.

```
data Pair a     = Pair a a
data TwoThree a = Node2 a a | Node3 a a a
```


5.7 Konstruktorklassen

Wie können nun Funktionen definiert werden, die den Typ `XTree` verwenden und in `sons` polymorph sind?

- Die Antwort läuft auf den Einsatz von so genannten *Konstruktorklassen* hinaus, die für eine Verallgemeinerung von Typklassen stehen.
- Während eine Typklasse eine Menge von Typen beschreibt, beschreibt eine *Konstruktorklasse eine Menge von Typkonstruktoren gleichen Kinds*.
- Ähnlich der Typklasse wird zunächst die Konstruktorklasse definiert und ein Name festgelegt. Anschliessend werden mögliche Exemplare für die Klasse eingeführt.

5.7 Konstruktorklassen

Aufgabe (Funktion `traverse`)

Es soll eine Funktion definiert werden, die zu einem Knoten `XBranch` eines Baumes die Liste der direkten Teilbäume liefert.

Zunächst wird eine Funktion benötigt, die Paare, Listen oder Werte aus `TwoThree` auf Listen abbildet.

Definition (Funktion `destruct` und `construct`)

```
destruct :: struct a -> [a]
```

`destruct` ist parametrisiert über die Struktur `struct` und muss daher als Methode einer Konstruktorklasse `Components` vereinbart werden. Die Alternative

```
construct :: [a] -> struct a
```

erstellt eine Struktur aus einer Liste.

5.7 Konstruktorklassen

```
-- Konstruktorklasse

class Components struct where
      construct :: [a] -> struct a
      destruct  :: struct a -> [a]

-- Exemplare

instance Components Pair where
         construct [x,y]      = Pair x y
         destruct  (Pair x y) = [x,y]
instance Components [] where
         construct xs = xs
         destruct  xs = xs
```


5.7 Konstruktorklassen

```
instance Components TwoThree where
         construct [x,y]   = Node2 x y
         construct [x,y,z] = Node3 x y z
         destruct  (Node2 x y)   = [x,y]
         destruct  (Node3 x y z) = [x,y,z]
```


5.7 Konstruktorklassen

Die Funktion zum Durchlaufen von beliebigen Bäumen kann nun ganz allgemein für *alle Typkonstruktoren* `struct` erklärt werden. Es handelt sich dabei um Typkonstruktoren, die Exemplare von `Components` sind.

```
traverse :: Components sons => XTree sons a -> [a]

traverse XMt             = [ ]
traverse (XBranch x xs) = x: concat (map traverse (destruct xs))

? traverse (XBranch 1 [XMt])
[1]::[Int]
? traverse (XBranch 1 (Pair XMt XMt))
[1]::[Int]
? traverse (XBranch 1 (Node3 (XBranch 2 (Node2 XMt XMt)) XMt XMt))
[1, 2]::[Int]
```