Funktionale Programmierung

Wiederholung

Andreas Abel und Steffen Jost (und Hans-Wolfgang Loidl)

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

17. April 2012

Preliminaries

Basistypen in Haskell:

- *Bool*: Boole'sche (logische) Werte: *True* und *False*
- *Char*: Zeichen
- *String*: Zeichenfolgen: Listen von Zeichen, d.h äquivalent zu [*Char*]
- *Int*: Ganze Zahlen ("fixed precision integers")
- *Integer*: Ganze Zahlen beliebiger Länge ("arbitrary precision integers")
- *Float*: Fließkommazahlen ("single-precision floating point numbers")

Preliminaries

Zusammengesetzte Typen:

- **Listen**: $[\cdot]$, z.B. $[Int]$ als Listen von Integers, mit Werten wie $[1, 2, 3]$
- **Tupel**: $(\cdot, \cdots)$, z.B. $(Bool, Int)$ als Tupel von booleschen Werten und Integers, mit Werten wie $(True, 2)$
- **Verbunde**: $\cdot \{ \cdot, \cdots \}$, z.B. $BI \{ b :: Bool, i :: Int \}$ mit Werten wie $BI \{b = True, i = 2\}$
- **Funktionen**: $a \to b$, z.B. $Int \to Bool$

Typsynonyme werden wie folgt definiert:

type $IntList = [Int]$

Typdeklarationen

Typdeklarationen haben folgendes Format: $e :: \tau$
d.h. "*Ausdruck e hat den Typ* τ."

Beispiele:

$[1, 2, 3] :: [Int]$
$x :: Bool$
$bi :: BI\ \{\ b :: Bool,\ i :: Int\ \}$
$inc :: Int \rightarrow Int$

Typdeklaration	
SML:	**Haskell:**
$val\ inc : int \rightarrow int$	$inc :: Int \rightarrow Int$

I. Algebraische Datentypen

Haskell stellte mächtige Mechanismen der Datenabstraktion zur Verfügung.
Algebraische Datentypen definieren benannte Alternativen ("constructors"), die beliebige Typen als Argumente haben können.

Ein Beispiel einer Enumeration, d.h. eines algebraischen Datentyps mit Alternativen ohne Argumente:

$$\textbf{data}\ \mathit{Day} = \mathit{Mon} \mid \mathit{Tue} \mid \mathit{Wed} \mid \mathit{Thu} \mid \mathit{Fri} \mid \mathit{Sat} \mid \mathit{Sun}$$

Algebraische Datentypen

Musterangleich ("**pattern matching**") über diesen Datentyp kann wie folgt verwendet werden:

$$
\begin{aligned}
&\mathit{next} && :: \mathit{Day} \to \mathit{Day} \\
&\mathit{next}\ \mathit{Mon} && = \mathit{Tue} \\
&\mathit{next}\ \mathit{Tue} && = \mathit{Wed} \\
&\mathit{next}\ \mathit{Wed} && = \mathit{Thu} \\
&\mathit{next}\ \mathit{Thu} && = \mathit{Fri} \\
&\mathit{next}\ \mathit{Fri} && = \mathit{Sat} \\
&\mathit{next}\ \mathit{Sat} && = \mathit{Sun} \\
&\mathit{next}\ \mathit{Sun} && = \mathit{Mon}
\end{aligned}
$$

Algebraische Datentypen

Ein weiteres Beispiel eines algeb. Datentyps sind komplexe Zahlen:

$$\textbf{data}\ \mathit{Complex1} = \mathit{Comp1}\ \mathit{Float}\ \mathit{Float}$$

Die folgende Funktion erzeugt eine komplexe Zahl aus Real- und Imaginärteil:

$$\begin{aligned} &\mathit{mkComp1} :: \mathit{Float} \to \mathit{Float} \to \mathit{Complex1} \\ &\mathit{mkComp1}\ r\ i = \mathit{Comp1}\ r\ i \end{aligned}$$

Die folgenden Funktionen extrahieren Real- und Imaginärkomponenten einer komplexen Zahl:

$$\begin{aligned} &\mathit{realPart1} :: \mathit{Complex1} \to \mathit{Float} \\ &\mathit{realPart1}\ (\mathit{Comp1}\ r\ \_) = r \\ &\mathit{imagPart1} :: \mathit{Complex1} \to \mathit{Float} \\ &\mathit{imagPart1}\ (\mathit{Comp1}\ \_\ i) = i \end{aligned}$$

Algebraische Datentypen

Algebraische Datentypen können **Typvariablen** als Parameter verwenden.
Als Beispiel können wir komplexe Zahlen mit einem Basistyp parametrisieren:

$$\textbf{data}\ \mathit{Complex2}\ a\ =\ \mathit{Comp2}\ a\ a$$

Die Funktionen sind **polymorph**, d.h. sie arbeiten für beliebige Instanzen der Typvariablen a

$$\begin{aligned}
&\mathit{mkComp2} &&:: a \to a \to \mathit{Complex2}\ a\\
&\mathit{mkComp2}\ r\ i &&= \mathit{Comp2}\ r\ i
\end{aligned}$$

$$\begin{aligned}
&\mathit{realPart2} &&:: \mathit{Complex2}\ a \to a\\
&\mathit{realPart2}\ (\mathit{Comp2}\ r\ _) &&= r\\
&\mathit{imagPart2} &&:: \mathit{Complex2}\ a \to a\\
&\mathit{imagPart2}\ (\mathit{Comp2}\ _\ i) &&= i
\end{aligned}$$

Algebraische Datentypen

Als verkürzende Schreibweise ist es möglich Infix-Konstrukturen zu verwenden (diese müssen mit einem : Symbol beginnen).

```
data Complex3 a = a :+ a


mkComp3     :: a -> a -> Complex3
mkComp3 r i = r :+ i


realPart3          :: Complex3 a -> a
realPart3 (r :+ _) = r

imagPart3          :: Complex3 a -> a
imagPart3 (_ :+ i) = i
```

Algebraische Datentypen

Wir können auch einen Verbund verwenden, um benannte Felder in der Datenstruktur zu erhalten.

$$\textbf{data}\ \mathit{Complex4}\ a\ =\ \mathit{Comp4}\ \{\ \mathit{real}\ ::\ a\ ,\ \mathit{imag}\ ::\ a\ \}$$

$$\begin{aligned}&\mathit{mkComp4}\ \ \ ::\ a\ \rightarrow\ a\ \rightarrow\ \mathit{Complex4}\ a\\&\mathit{mkComp4}\ r\ i\ =\ \mathit{Comp4}\ \{\ \mathit{real}\ =\ r,\ \mathit{imag}\ =\ i\ \}\end{aligned}$$

$$\begin{aligned}&\mathit{realPart4}\ \ \ ::\ \mathit{Complex4}\ a\ \rightarrow\ a\\&\mathit{realPart4}\ x\ =\ \mathit{real}\ x\end{aligned}$$

$$\begin{aligned}&\mathit{imagPart4}\ \ \ ::\ \mathit{Complex4}\ a\ \rightarrow\ a\\&\mathit{imagPart4}\ x\ =\ \mathit{imag}\ x\end{aligned}$$

Pattern Matching in Haskell

"Pattern matching" ist eine verkürzende Schreibweise für eine explizite Fallunterscheidung über den Typ.
Folgende Funktion erzeugt die **Liste der Quadrate** der Eingabeliste.

$$
\begin{aligned}
sqs &:: [Int] \to [Int] \\
sqs\ [] &= [] \\
sqs\ (x : xs) &= x * x\ :\ sqs\ xs
\end{aligned}
$$

Diese Definition ist äquivalent zu

$$
\begin{aligned}
sqs\ xs' = \ & \textbf{case}\ xs'\ \textbf{of} \\
& [] \to [] \\
& (x : xs) \to x * x\ :\ sqs\ xs
\end{aligned}
$$

Pattern Matching in Haskell

$$
\begin{aligned}
sqs &:: [Int] \to [Int] \\
sqs\ [] &= [] \\
sqs\ (x : xs) &= x * x\ :\ sqs\ xs
\end{aligned}
$$

Das allgemeine Schema von pattern matching code lässt sich aus folgender Schreibweise ablesen:

$$
\begin{aligned}
&sqs\ xs' = \\
&\quad \textbf{if}\ null\ xs'\ \textbf{then}\ [] \\
&\quad \textbf{else let} \\
&\qquad\qquad x = head\ xs' \\
&\qquad\qquad xs = tail\ xs' \\
&\qquad \textbf{in} \\
&\qquad x * x\ :\ sqs\ xs
\end{aligned}
$$

Notation: Layout rule

Um Elemente einer Sequenz von Definitionen oder Case-Zweigen zusammenzufassen, muss jedes Element mindestens so weit wie das erste Element eingerückt sein ("**layout**" rule).

Man kann auch die längere Notation $\{\ldots;\ldots\}$ verwenden, z.B:

$$
\begin{array}{l}
sqs\ xs' = \\
\quad \textbf{if}\ null\ xs'\ \textbf{then}\ [] \\
\quad \textbf{else let}\ \{\ x = head\ xs'\ ;\ xs = tail\ xs'\ \}\ \textbf{in}\ x * x\ :\ sqs\ xs
\end{array}
$$

II. Funktionen höherer Ordnung ("higher-order functions")

Viele Funktionen über Listen folgen einem generellen Format:

$$\begin{aligned} f\ [] &= v \\ f\ (x : xs) &= x \oplus f\ xs \end{aligned}$$

Zum Beispiel kann die Summe über eine Liste wie folgt definiert werden:

$$\begin{aligned} sum\ [] &= 0 \\ sum\ (x : xs) &= x + (sum\ xs) \end{aligned}$$

Die *foldr* Funktion

Die higher-order Funktion *foldr* abstrahiert diese Berechnungsstruktur

$$\begin{aligned} &\mathit{foldr} && :: (a \to b \to b) \to b \to [a] \to b \\ &\mathit{foldr}\ f\ v\ [] && = v \\ &\mathit{foldr}\ f\ v\ (x : xs) && = f\ x\ (\mathit{foldr}\ f\ v\ xs) \end{aligned}$$

Die *foldr* Funktion ersetzt jedes : in der Eingabeliste durch ein f, und jedes [] durch ein v:

Auswertung

$$\mathit{foldr}\ \oplus\ v\ (x_0 : \ldots : x_n : []) \Longrightarrow x_0 \oplus \ldots \oplus (x_n \oplus v)$$

Wir können nun die Summe über eine Liste wie folgt definieren:

$$\mathit{sum} = \mathit{foldr}\ (+)\ 0$$

Funktionen höherer Ordnung

Funktionen höherer Ordnung ("higher-order functions") nehmen Funktionen als Argumente oder liefern Funktionen als Resultate.

Im Allgemeinen können Funktionen wie Daten verwendet werden, z.B. als Elemente von Datenstrukturen ("*functions are first-class values*").

Funktionen höherer Ordnung werden benutzt, um häufig verwendete Berechnungsstrukturen zu abstrahieren.

Die *foldl* Funktion

Die higher-order Funktion *foldl* ist die links-assoziative Version von *foldr*:

$$
\begin{aligned}
&\mathit{foldl} && :: (b \to a \to b) \to b \to [a] \to b \\
&\mathit{foldl}\ f\ v\ [] && = v \\
&\mathit{foldl}\ f\ v\ (x : xs) && = \mathit{foldl}\ f\ (f\ v\ x)\ xs
\end{aligned}
$$

Auswertung

$$\mathit{foldl}\ \oplus\ v\ (x_0 : \ldots : x_n : []) \Longrightarrow (\ldots(v \oplus x_0) \oplus \ldots \oplus x_n)$$

Die *map* Funktion

Die vordefinierte, higher-order Funktion *map* wendet eine Funktion f auf alle Elemente der Eingabeliste an, d.h. es gilt

Auswertung

$$map\ f\ [x_0, x_1, \ldots, x_n] \Longrightarrow [f\ x_0,\ \ldots,\ f\ x_n]$$

Wir können nun unser Beispiel der Quadrate aller Zahlen in *xs* wie folgt definieren:

```
sqs     :: [Int] → [Int]
sqs xs =  map square xs
  where
    square x = x * x
```

Lambda Ausdrücke

Im vorangegangenen Beispiel haben wir die *square* Funktion nur einmal verwendet. In solchen Fällen ist es bequemer eine lokale Definition zu vermeiden, und statt dessen einen Lambda Ausdruck zu verwenden.
Ein **Lambda Ausdruck** ist eine anonyme Funktion, d.h. er definiert wie das Resultat in Abhängigkeit von der Eingabe berechnet wird ohne der Funktion einen Namen zu geben.

$$sqs :: [Int] \rightarrow [Int]$$
$$sqs = map\ (\lambda x \rightarrow x * x)$$

Lambda Ausdrücke	
SML:	**Haskell:**
`fn x => ...`	`\ x -> ...`

Funktionskomposition

Da Funktionen Werte erster Klasse sind, können sie auch miteinander kombiniert werden, z.B $(f\ .\ g)$ wobei folgendes gilt:

$$(f\ .\ g)\ x\ =\ f\ (g\ x)$$

Wir definieren die Liste der Quadrate aller geraden Zahlen:

$$sq\_even\ ::\ [Int]\ \rightarrow\ [Int]$$
$$sq\_even\ =\ map\ (\lambda x\ \rightarrow\ x * x)\ .\ filter\ even$$

Die Funktion *filter p xs* liefert alle Elemente der Liste *xs*, für die das Prädikat *p True* liefert.

Funktionskomposition	
SML:	**Haskell:**
`f o g`	`f . g`

Partielle Applikation

In Sprachen wie SML werden die Argumente von Funktionen meist zu Tupeln zusammengefasst:

$$\begin{aligned} add' &:: (Int, Int) \to Int \\ add'\,(x, y) &= x + y \end{aligned}$$

Funktionen, die ihre Argumente schrittweise verwenden, werden als "**curried functions**" bezeichnet, z.B:

$$\begin{aligned} add &:: Int \to (Int \to Int) \\ add\,x\,y &= x + y \end{aligned}$$

In dieser Version nimmt die *add* Funktion ein Argument und liefert eine Funktion $Int \to Int$ als Resultat. Der Ausdruck $add\ 1\ 2$ ist wie $(add\ 1)\ 2$ zu lesen. Man bezeichnet $add\ 1$ als eine **partielle Applikation**.

Partielle Applikation

Partielle Applikationen sind nützlich, wenn man Argumente, die sich selten ändern, fixieren will, z.B:

$$\begin{aligned} &inc :: Int \to Int \\ &inc = add\ 1 \end{aligned}$$

Um die Werte aller Listenelemente zu erhöhen definieren wir

$$\begin{aligned} &incAll :: [Int] \to [Int] \\ &incAll = map\ inc \end{aligned}$$

Wir können diese Definition auch kürzer schreiben:

$$\begin{aligned} &incAll :: [Int] \to [Int] \\ &incAll = map\ (+1) \end{aligned}$$

Durch das Weglassen eines Arguments eines binären Operators wird eine partielle Applikation definiert, z.B. ist der Ausdruck $(+1)$ äquivalent zu $\lambda x \to x + 1$.

Beispiel: Cesar Cipher

Das Prinzip von Caesar's Verschlüsselungsfunktion ("cesar cipher"):

Verschlüsselung (*encode*):
Gegeben: Eine Liste von Zeichen *xs*.
Gesucht: Eine Liste von Zeichen *encode n xs*, in der jedes *m*-te Zeichen des Alphabets durch das $m + n$-te Zeichen ersetzt ist.

Entschlüsselung (*crack*):
Gegeben: Eine Liste von Zeichen *xs*.
Gesucht: Eine Liste von Zeichen *crack xs*, sodass

$$\exists n.xs = encode\ n\ (crack\ xs).$$

Cesar Cipher: Hilfsfunktionen

Idee: Jedes Zeichen (Kleinbuchstabe) muss um einen gegebenen Betrag verschoben werden.

Hilfsfunktionen:

$$\begin{aligned} &let2int :: Char \to Int \\ &let2int\ c = ord\ c - ord\ 'a' \end{aligned}$$

$$\begin{aligned} &int2let :: Int \to Char \\ &int2let\ n = chr\ (ord\ 'a' + n) \end{aligned}$$

Cesar Cipher: Verschlüsselung

Verschiebefunktion:

$$\begin{aligned} &\mathit{shift} :: \mathit{Int} \to \mathit{Char} \to \mathit{Char} \\ &\mathit{shift}\ n\ c = \mathit{int2let}\ ((\mathit{let2int}\ c + n)\ `\mathit{mod}`\ 26) \end{aligned}$$

Verschlüsselungsfunktion:

$$\begin{aligned} &\mathit{encode} :: \mathit{Int} \to \mathit{String} \to \mathit{String} \\ &\mathit{encode}\ n\ cs = [\ \mathit{shift}\ n\ c \mid c \leftarrow cs\ ] \end{aligned}$$

Cesar Cipher: Entschlüsselung

Idee: Verschiebe den Eingabestring so, dass die Häufigkeiten der Zeichen möglichst nah an den bekannten Häufigkeiten von Kleinbuchstaben im Alphabet liegen.

Wir benötigen

- eine Tabelle der Häufigkeiten von Kleinbuchstaben im Alphabet;
- Hilfsfunktionen zum Ermitteln der Häufigkeiten und deren Distanz;
- eine Entschlüsselungsfunktion *crack*, gemäß obiger Idee.

Cesar Cipher: Entschlüsselung

Die bekannten Häufigkeiten von Kleinbuchstaben im Alphabet sind durch folgende Tabelle definiert:

$$
\begin{aligned}
\mathit{table} &:: [\mathit{Float}] \\
\mathit{table} &= [8.2,\ 1.5,\ 2.8,\ 4.3,\ 12.7,\ 2.2, \\
&\quad\ 2.0,\ 6.1,\ 7.0,\ 0.2,\ 0.8,\ 4.0,\ 2.4, \\
&\quad\ 6.7,\ 7.5,\ 1.9,\ 0.1,\ 6.0,\ 6.3, \\
&\quad\ 9.1,\ 2.8,\ 1.0,\ 2.4,\ 0.2,\ 2.0,\ 0.1]
\end{aligned}
$$

Cesar Cipher: Hilfsfunktionen

Berechnung des Prozentsatzes:

$$
\begin{aligned}
&\mathit{percent} &&:: \mathit{Int} \to \mathit{Int} \to \mathit{Float} \\
&\mathit{percent}\ n\ m &&= (\mathit{fromIntegral}\ n\ /\ \mathit{fromIntegral}\ m) * 100
\end{aligned}
$$

Berechnung der Häufigkeiten aller Kleinbuchstaben im String *cs*:

$$
\begin{aligned}
&\mathit{freqs} &&:: \mathit{String} \to [\mathit{Float}] \\
&\mathit{freqs}\ cs &&= [\mathit{percent}\ (\mathit{count}\ c\ cs)\ n \mid c \leftarrow ['a'..'z']\,] \\
&\quad \textbf{where} \\
&\qquad n &&= \mathit{lowers}\ cs
\end{aligned}
$$

Cesar Cipher: Hilfsfunktionen

Links-Rotation einer Liste: $rotate\ n\ xs\ =\ drop\ n\ xs\ +\!\!+\ take\ n\ xs$
Anzahl der Kleinbuchstaben in einem String:

$$\begin{aligned} lowers\ &::\ String\ \to\ Int \\ lowers\ cs\ &=\ length\ [\ c\ \mid\ c\ \leftarrow\ cs,\ isLower\ c] \end{aligned}$$

Häufigkeit eines Zeichens in einem String:

$$\begin{aligned} count\ &::\ Char\ \to\ String\ \to\ Int \\ count\ c\ cs\ &=\ length\ [\ c'\ \mid\ c'\ \leftarrow\ cs,\ c == c'] \end{aligned}$$

Auftreten eines Zeichens in einem String:

$$\begin{aligned} positions\ &::\ Eq\ a\ \Rightarrow\ a\ \to\ [a]\ \to\ [Int] \\ positions\ x\ xs\ &=\ [\ i'\ \mid\ (x',\ i')\ \leftarrow\ zip\ xs\ [0..n],\ x == x'\ ] \\ &\quad \textbf{where} \\ &\qquad n\ =\ length\ xs\ -\ 1 \end{aligned}$$

Cesar Cipher: Vergleich von Häufigkeiten

Zum Vergleich einer Sequenz von beobachteten Werten *os* mit einer Sequenz von erwarteten Werten *es* wird die *Chi-Quadrat* Funktion χ verwendet. Diese Funktion bestimmt die Distanz, d.h. je geringer der Wert ist desto näher sind die beobachteten Werte an den erwarteten Werten.

$$\chi\ os\ es\ = \sum_{i=0}^{n-1} \frac{(os_i - es_i)^2}{es_i}$$

Diese Funktion kann wie folgt implementiert werden:

$$\begin{aligned} &chisqr && :: [Float] \to [Float] \to Float \\ &chisqr\ os\ es = && sum\,[((o-e)\uparrow 2)/e \mid (o,e) \leftarrow zip\ os\ es] \end{aligned}$$

Cesar Cipher: Entschlüsselung

Die Entschlüsselungsfunktion bestimmt für jeden Verschiebungsfaktor *n* die *chisqr* Distanz, und wählt den Faktor *factor* mit der geringsten Distanz.

$$
\begin{aligned}
&\mathit{crack} \;::\; \mathit{String} \to \mathit{String}\\
&\mathit{crack}\ \mathit{cs} = \mathit{encode}\ (-\mathit{factor})\ \mathit{cs}\\
&\quad \textbf{where}\\
&\qquad \mathit{factor} = \mathit{head}\ (\mathit{positions}\ (\mathit{minimum}\ \mathit{chitab})\ \mathit{chitab})\\
&\qquad \mathit{chitab} = [\ \mathit{chisqr}\ (\mathit{rotate}\ n\ \mathit{table}')\ \mathit{table} \mid n \leftarrow [0..25]\ ]\\
&\qquad \mathit{table}' = \mathit{freqs}\ \mathit{cs}
\end{aligned}
$$