Einführung in die Funktionale Programmierung mit Haskell

Erste Schritte, Typen und Funktionen

Steffen Jost

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

17. April 2013

GHC

In der Vorlesung arbeiten wir mit der Haskell Platform, welche aus GHC, Bibliotheken und Dokumentation besteht. Die meiste Dokumentation ist aber auch online verfügbar, wie zum Beispiel
`http://www.haskell.org/ghc/docs/latest/html/libraries/`

Glasgow Haskell Kompiler (GHC) kennt zwei Arbeitsweisen:

- **GHC** Normaler Kompiler. Ein Programm wird in mehreren Dateien geschrieben und mithilfe des GHC in ein ausführbares Programm übersetzt.
- **GHCi** Interpreter Modus: Man gibt Definitionen ein und GHCI wertet diese sofort aus und zeigt den Wert an.

Wir befassen uns zunächst primär mit dem interaktiven Modus.

GHCi

```
> ghci

GHCi, version 7.4.2: http://www.haskell.org/ghc/
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> 1 + 2
3
Prelude> 3 + 4 * 5
23
Prelude> (3 + 4) * 5
35
Prelude>
```

Der Text hinter dem Prompt `Prelude>` wurde vom Benutzer getätigt, alles andere sind Ausgaben von GHCi. Der Prompt gibt per Default alle geladenen Bibliotheken (**Module**) an.

GHCi

Der Interpreter wertet alle eingegebenen Ausdrücke aus. Mit den Pfeiltasten kann von vorherige Eingaben durchblättern

Zur Steuerung des Interpreters stehen Befehle zur Verfügung, welche alle mit einem Doppelpunkt beginnen, z.B. `:?` für die Hilfe.

Alle Befehle kann man abkürzen. So kann man den Interpreter sowohl mit `:quit` also auch mit `:q` verlassen. Für uneindeutige Abkürzungen gibt es voreingestellte Defaults.

Auch für GHCi macht es Sinn, Programmdefinitionen mit einem gewöhnlichen Texteditor in eine separate Datei speichern und dann in den Interpreter zu laden, um nicht immer alles neu eintippen zu müssen.

```
:l datei.hs -- lade Definition aus Datei datei.hs
:r          -- erneut alle offenen Dateien einlesen
```

Ausdrücke & Werte

GHCi wertet Ausdrücke aus. Ein Ausdruck kann

- atomar sein, wie z.B. `1337`, `'a'`
- zusammengesetzt sein, wie z.B. `12+1`, `27 * (7 + 5)`

Ein Ausdruck kann also aus mehreren Unterausdrücken bestehen. Mit Klammern können wir explizit angeben, in welcher Reihenfolge ein Ausdruck aus seinen Unterausdrücken zusammengesetzt ist.

Ein Ausdruck hat (meistens) auch einen **Wert**; so ist 13 der Wert des Ausdrucks 12+1.

Jeder korrekt gebildete Ausdruck besitzt einen **Typ**.
Zum Beispiel: der Ausdruck 12+1 hat den Typ `Int`.
Wir schreiben kurz `12+1 :: Int`

Typen

Ein Typ ist eine Menge von Werten; so bezeichnet der Typ `Int` meistens die Menge der ganzen Zahlen von -2^{29} bis $2^{29} - 1$.

Typnamen beginnen in Haskell *immer* mit einem Großbuchstaben!

Mit dem Befehl `:t` kann man sich den Typ eines Ausdrucks anzeigen lassen:

```
Prelude> :t 'a'
'a' :: Char
```

Mit dem Befehl `:set +t` wird der Typ jedes ausgewerteten Ausdrucks angezeigt, mit `:unset +t` stellt man das wieder ab.

Wichtige Numerische Typen

Int Ganze Zahlen ("fixed precision integers"), maschinenabhängig, mindestens von -2^{29} bis $2^{29} - 1$. Es wird nicht auf Überläufe geprüft.

Integer Ganze Zahlen beliebiger Größe

"arbitrary precision integers"

Float Fließkommazahlen mindestens nach IEEE standard

"single-precision floating point numbers"

Double Fließkommazahlen mit mindestens doppelter Genauigkeit nach IEEE standard

"double-precision floating point numbers"

Rational Rationale Zahlen beliebiger Genauigkeit, werden mit dem Prozentzeichen konstruiert: `1 % 5` ≈ 0.2

'%' nicht im Prelude-Modul enthalten

Haskell kennt viele weitere numerische Datentypen, z.B. komplexe Zahlen, Uhrzeiten oder Festkommazahlen.

Wichtige Typen

Neben Zahlen gibt es noch weitere wichtige grundlegende Typen:

Bool Boole'sche (logische) Wahrheitswerte: `True` und `False`

Char Unicode Zeichen, z.B. `'q'`. Diese werden immer in Apostrophen eingeschlossen.

String Zeichenketten, z.B. `"Hallo!"`. Diese werden immmer in Anführungszeichen eingeschlossen.

Haskell ist so knapp wie nötig formuliert. Viele Dinge, welche in anderen Sprachen "speziell" oder "eingebaut" sind, werden in Haskell in einer gewöhnlichen Bibliothek definiert.

Von diesen drei Typen ist lediglich `Char` etwas grundlegender, die beiden anderen kann man leicht selbst definieren; doch dazu lernen wir später noch mehr.

Kartesisches Produkt

Ein Beispiel für einen **zusammengesetzten Typen** hat als Vorlage aus der Mathematik das kartesische Produkt:
Aus den Mengen A_1 und A_2 Mengen bildet man als **kartesisches Produkt** die Menge

$$A_1 \times A_2 = \{(a_1, a_2) \mid a_1 \in A_1 \text{ und } a_2 \in A_2\}$$

Beispiel: $(5, 8) \in \mathbb{N} \times \mathbb{N}$

Definition (Kartesisches Produkt)

Sind $A_1, \ldots A_n$ Mengen, so ist das kartesische Produkt definiert als $A_1 \times \cdots \times A_n = \{(a_1, \ldots, a_n) \mid a_i \in A_i \text{ für } i = 1 \ldots n\}$

Die Elemente von $A_1 \times, \ldots, \times A_n$ heißen allgemein ***n*-Tupel**, spezieller auch Paare, Tripel, Quadrupel, Quintupel, Sextupel,...

Tupel

In Haskell schreiben wir das genau so hin, lediglich im Typ verwenden wir anstelle des Symbols $\times$ ebenfalls Klammern und Kommas. Aus $\mathbb{Z} \times \mathbb{Z}$ wird also `(Integer,Integer)`:

```
Prelude> :set +t
Prelude> (7,6)
(7,6)
it :: (Integer, Integer)
Prelude> (42,'a')
(42,'a')
it :: (Integer, Char)
Prelude> (True,4.5,"Hi!")
(True,4.5,"Hi!")
it :: (Bool, Double, String)
Prelude>
```

Zusammengesetzten Typen und deren Werte können wir ganz genauso wie jeden anderen Typen verwenden.

Tupel

In Haskell schreiben wir das genau so hin, lediglich im Typ verwenden wir anstelle des Symbols $\times$ ebenfalls Klammern und Kommas. Aus $\mathbb{Z} \times \mathbb{Z}$ wird also `(Integer,Integer)`:

```
Prelude> :set +t
Prelude> (7,6)
(7,6)
it :: (Integer, Integer)
Prelude> (42,'a')
(42,'a')
it :: (Integer, Char)
Prelude> (True,4.5,"Hi!")
(True,4.5,"Hi!")
it :: (Bool, Double, [Char])
Prelude>
```

Zusammengesetzten Typen und deren Werte können wir ganz genauso wie jeden anderen Typen verwenden.

Listen

Einer der wichtigsten Typen in Haskell sind Listen, also geordnete Folgen von Werten. Listen schreiben wir in eckigen Klammern, wobei die Elemente durch Kommas getrennt werden:

```
[1,2,3] :: [Int]

[1,2,2,3,3,3] :: [Int]

["Hello","World","!"] :: [String]
```

Im Gegensatz zu einem Tupel wissen wir anhand des Typen nicht, wie viele Werte eine Liste enthält. Eine Liste kann sogar ganz leer sein, geschrieben `[]`.

Allerdings wissen wir, dass alle Elemente einer Liste den gleichen Typ haben — diesen schreiben wir ebenfalls in eckigen Klammern um den Listentyp hinzuschreiben; Listen sind daher *homogene* Datenstrukturen.

Listen

Listen und Tupel kann man beliebig ineinander verschachteln:

```
[(1,'a'),(2,'z'),(-4,'w')] :: [(Integer, Char)]

[[1,2,3],[],[4]] :: [[Integer]]

(4.5,[(True,'a',[5,7],())])
:: (Double, [(Bool, Char, [Integer], ())])
```

Achtung: `[]` und `[[]]` und `[[][]]` und `[[[]]]` und `[[],[[]]]` sind alles verschiedene Werte.

Das 0-Tupel ist ebenfalls erlaubt: `() :: ()`.
Der Typ `()` wird als **Unit**-Typ bezeichnet, und hat nur den einzigen Wert `()`. Aus dem Kontext wird fast immer klar, ob mit `()` der Typ oder der Wert gemeint ist.

Listen

Listen und Tupel kann man beliebig ineinander verschachteln:

```
[(1,'a'),(2,'z'),(-4,'w')] :: [(Integer, Char)]

[[1,2,3],[],[4]] :: [[Integer]]

(4.5,[(True,'a',[5,7],())])
:: (Double, [(Bool, Char, [Integer], ())])
```

Achtung: `[]` und `[[]]` und `[[][]]` und `[[[]]]` und `[[],[[]]]` sind alles verschiedene Werte.

Das 0-Tupel ist ebenfalls erlaubt: `() :: ()`.
Der Typ `()` wird als **Unit**-Typ bezeichnet, und hat nur den einzigen Wert `()`. Aus dem Kontext wird fast immer klar, ob mit `()` der Typ oder der Wert gemeint ist.

Listen

Listen und Tupel kann man beliebig ineinander verschachteln:

```
[(1,'a'),(2,'z'),(-4,'w')] :: [(Integer, Char)]

[[1,2,3],[],[4]] :: [[Integer]]

(4.5,[(True,'a',[5,7],())])
:: (Double, [(Bool, Char, [Integer], ())])
```

Achtung: `[]` und `[[]]` und `[[][]]` und `[[[]]]` und `[[],[[]]]` sind alles verschiedene Werte.

Das 0-Tupel ist ebenfalls erlaubt: `() :: ()`.
Der Typ `()` wird als **Unit**-Typ bezeichnet, und hat nur den einzigen Wert `()`. Aus dem Kontext wird fast immer klar, ob mit `()` der Typ oder der Wert gemeint ist.

Typabkürzungen

Der Typ `String` ist nur eine Abkürzung für eine `Char`-Liste:

```
type String = [Char]
```

Solche Abkürzungen darf man genau so auch selbst definieren: Hinter dem Schlüsselwort `type` schreibt meinen einen frischen Namen, der mit einem Großbuchstaben beginnt und hinter dem Gleichheitszeichen folgt ein bekannter Typ, z.B.

```
type MyWeirdType = (Double,[(Bool,Integer)])
```

Für die Ausführung von Programmen ist dies unerheblich. Typabkürzungen dienen primär zur Verbesserung der Lesbarkeit. Leider ignoriert GHC/GHCi Typabkürzungen meistens, d.h. GHCi gibt fast immer `[Char]` anstelle von `String` aus.

Es gibt noch weitere zusammengesetzte Typen, z.B. Records, Funktionstypen, etc.

Funktionen

Definition (Funktion)

Für zwei Mengen A und B ist eine (totale) **Funktion** f von A nach B eine Zuordnung, welche jedem Element $x \in A$ genau ein Element $y \in B$ zuordnet; geschrieben $f(x) = y$.
Die Menge A bezeichnen wir als den **Definitionsbereich** von f, die Menge B als **Zielbereich**.

Für eine Funktion f von A nach B schreiben wir kurz: $f : A \to B$.

Funktionsanwendung wird auch gerne ohne Klammer in **Präfixnotation** fx geschrieben, die **Postfixnotation** xf findet dagegen seltener Verwendung.

Beispiel: Funktionen

Totale Funktionen

- Minus : $\mathbb{Z} \to \mathbb{Z}$
- Nachfolgerfunktion $fx = x + 1 : \mathbb{N} \to \mathbb{N}$
- Signumfunktion

$$\text{sgn } arg = \begin{cases} +1 & \text{falls } x > 0 \\ 0 & \text{falls } x = 0 \\ -1 & \text{falls } x < 0 \end{cases} : \mathbb{R} \to \{-1, 0, +1\}$$

Partielle Funktionen

- Wurzelfunktion $\sqrt{\cdot} : \mathbb{R} \to \mathbb{R}$ mit Definitionsbereich $\mathbb{R}^+$
- Kehrwert $\texttt{kw}\ x = 1/x : \mathbb{R} \to \mathbb{R}$ mit Definitionsbereich $\mathbb{R} \setminus \{0\}$
- $\texttt{foo}\ x = \begin{cases} 2x & x > 1 \\ 0 & x < 1 \end{cases} : \mathbb{N} \to \mathbb{N}$

  mit Definitionsbereich $\mathbb{N} \setminus \{-1, 0, +1\}$

Partielle Funktionen

Definition (Partielle Funktion)

Eine **partielle Funktion** $f : A \to B$ ordnet nur einer Teilmenge $A' \subset A$ einen Wert aus B zu, und ist ansonsten undefiniert.
In diesem Falle bezeichnen wir A als **Quellbereich** und A' als Definitionsbereich.

Ob eine Funktion partiell oder total ist, kann man nicht immer leicht feststellen.

Wir sprechen von partiellen Funktionen, wenn bei der Berechnung einer Funktion für ein bestimmtes Argument ein Fehler auftritt

Beispiel: Funktionen

Totale Funktionen

- Minus : $\mathbb{Z} \to \mathbb{Z}$
- Nachfolgerfunktion $fx = x + 1 : \mathbb{N} \to \mathbb{N}$
- Signumfunktion

$$\mathtt{sgn}\ arg = \begin{cases} +1 & \text{falls } x > 0 \\ 0 & \text{falls } x = 0 \\ -1 & \text{falls } x < 0 \end{cases} : \mathbb{R} \to \{-1, 0, +1\}$$

Partielle Funktionen

- Wurzelfunktion $\sqrt{\cdot} : \mathbb{R} \to \mathbb{R}$ mit Definitionsbereich $\mathbb{R}^+$
- Kehrwert $\mathtt{kw}\ x = 1/x : \mathbb{R} \to \mathbb{R}$ mit Definitionsbereich $\mathbb{R} \setminus \{0\}$
- $\mathtt{foo}\ x = \begin{cases} 2x & x > 1 \\ 0 & x < 1 \end{cases} : \mathbb{N} \to \mathbb{N}$

  mit Definitionsbereich $\mathbb{N} \setminus \{-1, 0, +1\}$

Anonyme Funktionen

Merkwürdig: $\sqrt{\cdot} :: \mathbb{R} \to \mathbb{R}$ Was bedeutet der Punkt?

Es ist oft praktisch, simplen Funktionen keinen besonderen Namen zu geben, bzw. die Wurzelfunktion hat ja bereits ein eigenes Symbol.

Aus dem Lambda-Kalkül (Alonzo Church, 1936), der mathematischen Grundlage der funktionalen Programmierung, nehmen wir daher die λ-Notation für anonyme Funktionen:

Nachfolgerfunktion	$\lambda x \,.\, x + 1$
Wurzelfunktion	$\lambda y \,.\, \sqrt{y}$

In Haskell schreiben wir anstelle von λ einfach einen Rückstrich \

```
\x -> x + 1
\y -> sqrt y
```

Funktionen mit mehreren Argumenten

Eine Funktion, deren Quellbereich ein n-stelliges kartesisches Produkt ist, nennt man eine Funktion der **Stelligkeit** n, oder auch n-**stellige** Funktion "n-ary function", "arity".

$$f :: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$
$$f(x, y) = x + 2y^2$$

Für die Anwendung zweistelliger Funktionen wird öfters die **Infixnotation** verwendet: *xfy*

Bekannte Beispiele für zweistellige Funktionen mit gebräuchlicher Infixnotation sind $+, -, *, /$.

Funktionen mit mehreren Argumenten

Eine Funktion, deren Quellbereich ein n-stelliges kartesisches Produkt ist, nennt man eine Funktion der **Stelligkeit** n, oder auch n-**stellige** Funktion "n-ary function", "arity".

$$f :: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$
$$f(x, y) = x + 2y^2$$

Für die Anwendung zweistelliger Funktionen wird öfters die **Infixnotation** verwendet: xfy

Bekannte Beispiele für zweistellige Funktionen mit gebräuchlicher Infixnotation sind $+, -, *, /$.

Funktionen mit mehreren Argumenten

Eine Funktion, deren Quellbereich ein n-stelliges kartesisches Produkt ist, nennt man eine Funktion der **Stelligkeit** n, oder auch n-**stellige** Funktion "n-ary function", "arity".

$$f :: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$
$$f(x, y) = x + 2y^2$$

Für die Anwendung zweistelliger Funktionen wird öfters die **Infixnotation** verwendet: xfy

Bekannte Beispiele für zweistellige Funktionen mit gebräuchlicher Infixnotation sind $+, -, *, /$.

Funktionen in Haskell

In Haskell schreiben wir Funktion ganz natürlich hin:

```
succ :: Int -> Int
succ x = x + 1
```

Wir definieren eine Funktion mit dem Namen `succ`, welche eine ganze Zahl auf Ihren Nachfolger abbildet.
Den Namen des Argumentes, hier `x`, können wir frei wählen.
"Sprechende" Namen empfehlen sich.

Die erste Zeile, also die Angabe der **Typsignatur** der Funktion ist optional, da GHC den Typ auch automatisch inferieren kann.
Man sollte immer explizite Typsignaturen hinschreiben, denn...

- ...Typsignaturen sind eine hilfreiche Dokumentation
- ...helfen dabei, Fehlermeldungen leserlicher zu machen
- ...können Programme manchmal effizienter machen

Mehrstellige Funktionen

Mehrstellige Funktionen können wir auf zwei Arten definieren:

1-stellige Funktion mit Kartesischem Produkt

```
foo :: (Int,Int) -> Int
foo (x,y) = (x + 1) * y
```

Hier verwenden wir ganz einfach Tupel

Echte n-stellige Funktion

```
bar :: Int -> (Int -> Int)
bar x y = (x + 1) * y
```

Hier schreiben wir einfach alle Argumente durch Leerzeichen getrennt hintereinander.

Die zweite Variante wird oft bevorzugt; doch da eine Funktion immer nur *einen einzigen* Wert zurück liefern kann, bietet es sich manchmal an, Tupel zu verwenden.

Man kann auch leicht wechseln, Stichwort “Currying”.

Funktionsanwendung

Die Funktionsanwendung verwendet in Haskell primär die Präfixnotation, d.h. man schreibt hinter dem Namen der Funktion einfach alle Argumente, durch Leerzeichen getrennt:

```
Prelude>  succ 5
6

Prelude> foo (2,7)
21

Prelude> bar 1 8
16
```

Dies kann man auch mithilfe von Klammern verschachteln:

```
Prelude> bar (succ 5) 2
14
```

Infixnotation in Haskell

Haskell bietet ebenfalls die Infixnotationen an, um die Lesbarkeit des Codes zu erhöhen.
Der Programmier gibt bei der Funktionsdefinition an, welche Notation er bevorzugt:

```
infixl 6  +
(+) :: Int -> Int
(+) x

succ :: Int -> Int
succ x = x + 1
```

`infixl` oder `infixr` und die Zahl dahinter zeigen Haskell, in welcher Reihenfolge mehrere Unterausdrücke ohne explizite Klammerung zu lesen sind.
So wird zum Beispiel die Punkt-vor-Strich Regel eingehalten:

```
infixl 7  *
```

Infixnotation in Haskell

Zwischen Infix- und Präfixnotation kann man jederzeit auch ad hoc wechseln:

- Mit Klammern macht man aus einer Infix-Funktion eine gewöhnliche Präfix-Funktion:

```
Prelude> (+) 1 2
3
```

- Mit Rückstrichen "backquotes" macht man aus einer Funktion in Präfixnotation eine Infix-Funktion:

```
Prelude> 1 `bar` 8
16
```

Man verwendet die Notation, welche die Lesbarkeit erhöht.

Lesbarkeit des Codes ist Haskell-Programmieren sehr wichtig! Wir werden noch weiteren Mechanismen dazu begegnen.

Funktionstypen

Der Typ einer Funktion ist ein zusammengesetzter Funktionstyp, der immer aus genau zwei Typen mit einem Pfeil dazwischen besteht.

Funktionstypen sind implizit rechtsgeklammert, d.h. man darf die Klammern manchmal weglassen:

`Int -> Int -> Int` wird gelesen als `Int -> (Int -> Int)`

Entsprechend ist die Funktionsanwendung implizit linksgeklammert:

`bar 1 8` wird gelesen als `(bar 1) 8`

Das bedeutet: `(bar 1)` ist eine Funktion des Typs `Int -> Int`!
Funktionen sind also normale Werte in einer funktionalen Sprache!