Funktionen und Listenkonstruktion

Prof. Dr. Johannes Kinder

Ludwig-Maximilians-Universität München, Institut für Informatik

Lehrstuhl für Programmiersprachen und Künstliche Intelligenz

Programmierung und Modellierung, SoSe 2024

Zusammenfassung der letzten Vorlesung

- Grundlagen Funktionale Programmierung
- Basistypen — `Bool`, `Int`, `Char`, `Double`
- Kartesische Produkte — `(1,'a') :: (Int, Char)`
- Listen — `['H','i','!'] :: [Char]`
- Typabkürzung mit `type` — `type String = [Char]`

Funktionen

Definition (Funktion)

Für zwei Mengen A und B ist eine (totale) Funktion f von A nach B, geschrieben $f : A \to B$, eine Zuordnung, welche jedem Element $x \in A$ genau ein Element $y \in B$ zuordnet, geschrieben $f(x) = y$.

Die Menge A ist der **Definitionsbereich** von f, die Menge B der **Zielbereich**

- Funktionsanwendung wird oft auch ohne Klammer in **Präfixnotation** f x geschrieben, selten auch in **Postfixnotation** x f
- Den Definitionsbereich einer Funktion f (eng. Domain) bezeichnen wir üblicherweise mit $\mathrm{dom}(f)$

Beispiel: Funktionen

- Totale Funktionen
  - Nachfolgerfunktion $f\,x = x + 1 : \mathbb{N} \rightarrow \mathbb{N}$
  - Signumfunktion $\mathrm{sgn} = \begin{cases} +1 & \text{falls } x > 0 \\ 0 & \text{falls } x = 0 \\ -1 & \text{falls } x < 0 \end{cases} : \mathbb{R} \rightarrow \{-1, 0, +1\}$

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 Fall 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
  - Nachfolgerfunktion $f\,x = x + 1 : \mathbb{N} \to \mathbb{N}$
  - Signumfunktion $\operatorname{sgn} x = \begin{cases} +1 & \text{falls } x > 0 \\ 0 & \text{falls } x = 0 \\ -1 & \text{falls } x < 0 \end{cases} : \mathbb{R} \to \{-1, 0, +1\}$
- Partielle Funktionen
  - Wurzelfunktion $\sqrt{\cdot} : \mathbb{R} \to \mathbb{R}$ mit Definitionsbereich $\mathbb{R}^+$
  - Kehrwert $\mathrm{kw}\,x = \dfrac{1}{x} : \mathbb{R} \to \mathbb{R}$ mit Definitionsbereich $\mathbb{R}\backslash\{0\}$
  - $\mathrm{foo}\,x = \begin{cases} 2x & \text{falls } x > 1 \\ 0 & \text{falls } x < 1 \end{cases} : \mathbb{N} \to \mathbb{N}$ mit Definitionsbereich $\mathbb{N}\backslash\{1\}$

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 (engl. *n-ary function*, bzw. *of arity n*)

$$f :: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

$$f(x, y) = x + 2y^2$$

- Für die Anwendung zweistelliger Funktion wird oft die **Infixnotation** verwendet:

$$x\, f\, y$$

- Beispiele mit gebräuchlicher Infixnotation sind +, −, $*$, /

Funktionen in Haskell

- In Haskell werden Funktionen ähnlich wie in der Mathematik definiert:

```
succ :: Integer -> Integer

succ x = x + 1
```

- Definiert eine Funktion mit Namen `succ`, welche eine ganze Zahl auf ihren Nachfolger abbildet. Der Name des Arguments ist frei wählbar
- Die Angabe der **Typsignatur** in der ersten Zeile ist optional, da GHC den Typ auch automatisch inferieren kann
  - Explizite Typsignaturen dienen aber als hilfreiche Dokumentation;
  - helfen, Fehler in der Implementierung schneller zu entdecken (inferierter Typ passt nicht zu Typsignatur);
  - und helfen auch dabei, Fehlermeldungen leserlicher zu machen

Mehrstellige Funktionen in Haskell

- Einstellige Funktion mit Kartesischem Produkt (Argument ist ein Tupel)

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

- Echte n-Stellige Funktion (Argumente durch Leerzeichen getrennt)

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

- Zweite Variante wird für mehrere Argumente bevorzugt
  - Es lässt sich aber auch leicht wechseln, Stichwort „Currying"
- Da eine Funktion immer nur einen einzigen Wert zurückgeben kann, bieten sich für mehrere Rückgabewerte Tupel an

Funktionsanwendung

- Funktionsanwendung verwendet in Haskell primär die Präfixnotation

```
ghci> succ 5
6
ghci> foo (2,7)
21
ghci> bar 1 8
16
```

- Lässt sich auch mit Hilfe von Klammern verschachteln

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

Infixnotation in Haskell

- Haskell bietet ebenfalls Infixnotation an, um die Lesbarkeit zu erhöhen
  - Bei Funktionsdefinition angeben, welche Notation bevorzugt wird

```
infixl 6 +
(+) :: Integer -> Integer -> Integer
```

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

```
infixl 7 *
```

Wechsel zu Infixnotation

- Zwischen Infix- und Präfixnotation kann man im Code jederzeit wechseln:
  - Mit Klammern macht man aus einer Infix-Funktion eine gewöhnliche Präfix-Funktion:

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

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

```
ghci> (+) 1 `bar` 8
16
```

- Man verwendet die Notation, welche die Lesbarkeit erhöht.

Funktionstypen

- Der Typ einer Funktion ist ein zusammengesetzter **Funktionstyp**, der immer aus genau zwei Typen mit einem Pfeil dazwischen besteht
- Jede Funktion hat genau **ein Argument** und **ein Ergebnis**
- Klammerkonvention:
  - Funktionstypen sind implizit rechtsgeklammert, d.h. man darf die Klammern manchmal weglassen

    `Integer -> Integer -> Integer` wird gelesen als `Integer -> (Integer -> Integer)`
  - Entsprechend ist die Funktionsanwendung implizit linksgeklammert:

    `bar 1 8` wird gelesen als `(bar 1) 8`
- Das bedeutet: `(bar 1)` ist eine Funktion des Typs `Integer -> Integer`! Funktionen sind also normale Werte in einer funktionalen Sprache

Listenkonstruktion

- *Erinnerung*: Eine Liste ist eine geordnete Folge von Werten des gleichen Typs, mit beliebiger Länge, insbesondere auch Länge 0.
- Syntax für vollständige Listen: `[1,2,3,4,5]`
- Aufzählungen: `[1..5]`
- Mit Schrittweite: `[1,3,..10] == [1,3,5,7,9]`
  - Haskell Syntax `[a,b..m]` ist eine Abkürzung für die Liste $[a, a+1(b-a), 2(b-a), \ \ldots, a+n(b-a)]$, wobei n die größte natürliche Zahl mit $a+n(b-a) \leq m$ ist
  - Funktioniert mit allen „aufzählbaren" Typen (⇨Typklassen)
- List-Comprehension
- Infix Listenoperator (`:`)

List-Comprehension

- Mengen werden in der Mathematik of intensional beschrieben:

$$\left\{x^2 \mid x \in \{1,2,\ldots,10\} \text{ und } x \text{ ist ungerade}\right\} = \left\{1, 9, 25, 49, 81\right\}$$

  wird gelesen als „Menge aller x^2, so dass gilt…"

- Haskell bietet diese Notation ganz analog für Listen:

```
[ x^2 | x <- [1..10], odd x ] == [1,9,25,49,81]
```

  „Liste aller x^2, wobei x aus der Liste [1,…,10] gezogen wird und x ungerade ist"

- Es gibt auch echte ungeordnete Mengen in Haskell, aber Listen sind grundlegender

Struktur einer List-Comprehension

```
[ x^2 | x <- [1..10], odd x ] == [1,9,25,49,81]
```

- **Rumpf**: Bestimmt, wie ein Listenelement berechnet wird
- **Generator**: weist Variablen nacheinander Elemente einer anderen Liste zu hier die Liste `[1..10]`
- **Filter**: Ausdruck von Typ `Bool` (Bedingung) entscheidet, ob dieser Wert in erzeugter Liste enthalten ist
- **Abkürzung**: `let` erlaubt Abkürzungen zur Wiederverwendung

```
[ z | x <- [1..10], let z = x^2, z > 50] == [64, 81, 100]
```

- Beliebig viele Generatoren, Filter und Abkürzungen
- Definitionen können „weiter rechts" und im Rumpf verwendet werden
- List-Comprehensions können auch verschachtelt werden

Beispiele

- Beliebig viele Generatoren, Filter und Abkürzungen dürfen in beliebiger Reihenfolge in List-Comprehensions verwendet werden:

```
ghci> [ (wert,name) | wert <-[1..3], name <- ['a','b']]
[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]
```

- Die Reihenfolge der Generatoren bestimmt die Reihenfolge der Werte in der Ergebnisliste.

```
ghci> [ (wert,name) | name <- ['a','b'], wert <- [1..3]]
[(1,'a'),(2,'a'),(3,'a'),(1,'b'),(2,'b'),(3,'b')]
```

Listenkonstruktion

- *Erinnerung*: Eine Liste ist eine geordnete Folge von Werten des gleichen Typs, mit beliebiger Länge, insbesondere auch Länge 0.
- Syntax für vollständige Listen: `[1,2,3,4,5]`
- Aufzählungen: `[1..5]`
- Mit Schrittweite: `[1,3,..10] == [1,3,5,7,9]`
  - Haskell Syntax `[a,b..m]` ist eine Abkürzung für die Liste $[a, a+1(b-a), 2(b-a), \ldots, a+n(b-a)]$, wobei n die größte natürliche Zahl mit $a+n(b-a) \leq m$ ist
  - Funktioniert mit allen „aufzählbaren" Typen (⇨Typklassen)
- List-Comprehension
- **Infix Listenoperator** (`:`)

Listenkonstruktor Cons

- Jede nicht-leere Liste besteht aus einem **Kopf** und einem **Rumpf** (engl.: *head / tail*)
- Einer Liste kann man mit dem Infixoperator `(:)` ein neuer Kopf gegeben werden, der vorherige Kopf wird zum zweiten Element der Liste

```
ghci> 0:[1,2,3]
[0,1,2,3]
```

```
ghci> 'a':('b':['c'])
"abc"
```

- Tatsächlich ist `[1,2,3]` nur eine andere Schreibweise für `1:2:3:[]`, beide Ausdrücke sind äquivalent
- `(:)` ist rechtsassoziativ
- `(:)` konstruiert einen neuen Listenknoten, deshalb oft auch „Cons"-Operator genannt

Typvariablen

- Welchen Typ hat `(:)`?

```
ghci> :t (:)
(:) :: a -> [a] -> [a]
```

- Typen werden in Haskell immer groß geschrieben, `a` ist kein Typ!
- `a` ist eine **Typvariable** und steht für einen beliebigen Typen. Typvariablen werden immer klein geschrieben.
- Der Cons-Operator funktioniert also mit Listen beliebigen Typs
  - Der Ausdruck `'a':[1]` ergibt aber einen Typfehler, da `a` in demselben Ausdruck verschiedene Typen haben müsste!

Zusammenfassung

- Funktionen in Haskell haben ein Argument und ein Ergebnis
- Mehrstellige Funktionen:
  - Argument ist ein Produkt von n Argumenten als n-Tupel oder
  - Ergebnis ist eine Funktion, welche das nächste Argument verarbeitet (bevorzugt)
- Listenkonstruktion
  - Listen Aufzählungen `[1,3..99]`
  - List-Comprehensions `[x | x <- [1..9], even x]`
  - Cons-Constructor `(:) :: a -> [a] -> [a]`