Funktionales Programmieren

Teil 1

Carl Philipp Reh

Universität Siegen

24. November 2023

Übersicht

Ziele der Vorlesung:

- Einführung in Haskell.
- Wichtige Typklassen von Haskell.
- Denotationelle Semantik für (eine Teilsprache von) Haskell.
- Typüberprüfung.

Literatur:

- Funktionale Programmierung, Jürgen Giesl, `https://verify.rwth-aachen.de/fp19/FP19.pdf`

Länge einer Liste in C

```
struct element {
  struct data value;
  struct element *next;
};
struct list { struct element *head; };

size_t length(struct list *l) {
  size_t r = 0;
  struct element *cur = l->head;
  while(cur != NULL) {
    cur = cur->next;
    ++r;
  }
  return r;
}
```

Länge einer Liste in Haskell

```
len :: [a] -> Int
len l = case l of
    [] -> 0
    (x : xs) -> 1 + len xs
```

Länge einer Liste in Haskell

```
len :: [a] -> Int
len l = case l of
    [] -> 0
    (x : xs) -> 1 + len xs
```

Die erste Zeile ist eine *Typdeklaration* und sagt, dass `len` eine Funktion ist, die ein `[a]` als Argument bekommt und einen `Int` als Ergebnis liefert.

`[a]` bedeutet „Liste vom Typ `a`“, wobei `a` eine Typvariable ist, die jeden Typ annehmen kann, zum Beispiel `a=Int` oder `a=Bool`.

Länge einer Liste in Haskell

```
len :: [a] -> Int
len l = case l of
    [] -> 0
    (x : xs) -> 1 + len xs
```

Die Definition von Listen besteht aus zwei Fällen:

- `[]` ist die leere Liste, die jeden Typ annehmen kann, also `[] :: [t]` für jeden Typ `t`.
- Wenn `x :: t` und `xs :: [t]`, dann ist `(x : xs) :: [t]` die Liste, die man erhält, wenn man `x` vorne an `xs` hängt.

Zum Beispiel gilt `(5 : (3 : [])) :: [Int]`, was die Liste $(5, 3)$ ist.

Länge einer Liste in Haskell

```
len :: [a] -> Int
len l = case l of
    [] -> 0
    (x : xs) -> 1 + len xs
```

Die Definition von `len` muss die beiden Fälle (leere und nicht leere Liste) unterscheiden. Das geschieht mit **`case`**.

- Wenn die Liste leer ist, geben wir `0` zurück.
- Wenn die Liste nicht leer ist, dann geben wir `1 + len xs` zurück.

Länge einer Liste in Haskell

```
len :: [a] -> Int
len l = case l of
    [] -> 0
    (x : xs) -> 1 + len xs
```

Beispielauswertungen:

```
len [] = 0
```

```
len ("x" : ("y" : []))
  = 1 + (len ("y" : []))
  = 1 + (1 + len [])
  = 1 + (1 + 0)
  = 2
```

Länge einer Liste in Haskell

Weitere Bemerkungen:

- `+` ist in Haskell vordefiniert und ist hier die Addition auf **`Int`**.
- **`Int`** ist ein Typ mit positiven und negativen Zahlen, normalerweise 64 Bit groß.
- `"x"` ist ein String mit dem einzigen Symbol `'x'`.
- Listen können nur aus Elementen eines Typs bestehen. Beispielsweise ist `3 : ("x" : [])` ein Typfehler.
- Statt `(x_1 : ...(x_n : []) ...)` kann man auch `[x_1,...,x_n]` schreiben. Zum Beispiel ist `[1,2] = 1 : (2 : [])`.
- Haskell ist „whitespace-sensitiv“. D.h., die Einrückung in der Definition von `len` ist wichtig.

Länge einer Liste in Haskell

Fundamentale Unterschiede zwischen dem Haskell- und dem C-Programm:

- Keine Schleifen, nur Rekursion.
- Polymorphes Typsystem: Code funktioniert mit jeder Liste.
- Keine Seiteneffekte: Wir verändern nicht den Inhalt von Variablen.
- Automatische Speicherverwaltung: In dem C-Programm müsste man Werte vom Typ `element` mit `malloc` erzeugen und mit `free` freigeben. In Haskell werden die Listen, die wir verwenden, automatisch freigegeben.

Funktionen mit mehreren Argumenten

```
add :: Int -> Int -> Int
add x y = x + y
```

ist eine Funktion, die zwei Argumente erhält. Bei mehreren `->` sind Klammern optional, da implizit rechts geklammert wird. Das heißt, man könnte auch schreiben `add :: Int -> (Int -> Int)`. Dies bedeutet, dass `add` eine Funktion ist, die einen `Int` als Argument erhält und eine *Funktion liefert*, die noch einen `Int` als Argument erhält.

Funktionen mit mehreren Argumenten

Zum Beispiel kann man das Addieren von `5` definieren als

```
add5 :: Int -> Int
add5 = add 5
```

Auch beim Aufrufen von Funktionen werden in der Regel Klammern weggelassen. So bedeutet `add 1 2` dasselbe wie `(add 1) 2`.
Zum Beispiel gilt

```
add5 10 = (add 5) 10 =  add 5 10 = 5 + 10.
```

Funktionen höherer Ordnung

Eine Funktion, die eine andere Funktion als Argument erhält, nennt man *Funktion höherer Ordnung*. Einfaches Beispiel:

```
twice :: (a -> a) -> (a -> a)
twice f x = f (f x)
```

Funktionen nennt man daher auch „first-class“ in Haskell. D.h., sie können wie normale Daten auch an Funktionen übergeben werden. Beispiel:

```
add10 :: Int -> Int
add10 = twice add5
```

Dann erhalten wir zum Beispiel

```
add10 x = twice add5 x = add5 (add5 x).
```