5 Funktionale Programmiersprachen

5.1 Motivation

Imperative Programmiersprachen basieren auf der *Zuweisung* als einfachste Anweisung. Da jede Zuweisung ein *Seiteneffekt* auf Objektwerten ist, kann man die imperative Programmierung auch als „*seiteneffektbasierte Programmierung*“ interpretieren. Dies führt häufig zu Problemen:

- Die Effekte von Programmeinheiten (z. B. Prozeduren, Funktionen) sind schwer kontrollierbar.
- Es können viele kleine Fehler durch falsche Reihenfolgen (`i++` statt `++i`, ...) passieren.
- Die Auswertungsreihenfolge ist auch im Detail sehr relevant.

Beispiel 5.1 (Auswertungsreihenfolge in C)**.**

```
x = 3;
y = (++x) * (x--);
```

Hier ist das Ergebnis von `y` abhängig von der Auswertungsreihenfolge des `*`-Operators.

Beispiel 5.2 (Typische Fehlerquellen in imperativen Sprachen)**.** Wir betrachten eine imperative Implementierung der Fakultätsfunktion.

```
int fac(int n) {
  int z = 1;
  int p = 1;
  while (z < n + 1) {
    p = p * z;
    z++;
  }
  return p;
}
```

Schon bei diesem kleinen Programm gibt es viele mögliche Fehlerquellen:

- Initialisierung: `z = 0` oder `z = 1`? Ebenso für `p`?

- Abbruchbedingung: `z < n + 1` oder `z < n` oder `z <= n`?
- Reihenfolge bei Zuweisungen: `z++` hinter oder vor `p = p * z`?

Dagegen ist die mathematische Definition wesentlich klarer und daher weniger fehleranfällig:

$$fac(0) = 1$$
$$fac(n) = n * fac(n-1) \quad \text{falls } n > 0$$

Neben diesen vielen kleinen Fehlerquellen existiert bei der imperativen und insbesondere auch bei der objektorientierten Programmierung ein grundsätzliches Problem, welches man als „Nicht-Lokalität“ bezeichnen kann. Bei der mathematischen Definition der Fakultätsfunktion ist das Ergebnis nur abhängig von den Eingabeparametern, wodurch das Verständnis der Bedeutung dieser Funktion einfach ist.
Dagegen können bei imperativen Sprachen in jeder Funktion, Prozedur oder Methode Variablen benutzt und auch verändert werden, die außerhalb definiert sind. Dadurch ist das Ergebnis oder Verhalten nicht mehr nur durch eine einfache lokale Codeinspektion ersichtlich, sondern es müssen auch die globalen Variablen und andere Programmteile, die diese eventuell verändern, einbezogen werden. Bei der objektorientierten Programmierung kommt noch erschwerend das Prinzip der Vererbung hinzu. Zum Verständnis einer Klasse muss man nicht nur die Klasse selbst, sondern alle Klassen in der Vererbungshierarchie dieser Klasse berücksichtigten. Wenn man dann noch bedenkt, dass Klassen über Attribute noch auf die Funktionalität anderer Klassen verweisen können, wir das folgende Zitat von Joe Armstrong, dem Erfinder der Programmiersprache Erlang, verständlich:

> The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

Zu berücksichtigen ist, dass es nicht nur um das Verständnis, sondern auch um die Wartbarkeit von Software geht. Jeder Veränderung der Funktionalität einer Teilfunktion kann in imperativen und objektorientierten Programmiersprachen schwer überschaubare Änderungen an der Funktionalität davon direkt oder indirekt abhängiger Programmteile haben.

Um dieses Problem zu umgehen, basieren **funktionale Programmiersprachen** auf einem einfachen Prinzip: das Ergebnis einer Funktion ist nur abhängig von den Eingabeargumenten für diese Funktion und nicht von einem impliziten Zustand oder anderen zeitlich veränderbaren Größen. Somit kann das wichtigste Prinzip rein funktionaler Programmiersprachen wie folgt zusammengefasst werden:

> **Der Wert eines Ausdrucks hängt nur von den Werten seiner Teilausdrücke und nicht vom Zeitpunkt der Auswertung ab!**

Dies wird auch als **referenzielle Transparenz** bezeichnet. Es wird durch die folgenden Charakteristika funktionaler Programmiersprachen erreicht:

- Es gibt keine Zuweisung und damit keine Seiteneffekte.
- Es gibt keine Anweisungen oder Prozeduren, sondern nur Ausdrücke und Funktionen.
- Außerdem sind Funktionen „Werte 1. Klasse “ (first class values/citizens), d. h. sie können selbst Argumente oder Ergebnisse von Funktionen sein ($\rightsquigarrow$ Funktionen höherer Ordnung).

Einige praktische Konsequenzen dieses Prinzips sind:

- Der Wert eines Ausdrucks wie `x+y` hängt nur von den Werten von `x` bzw. `y` ab.
- Variablenänderungen in Ausdrücken, wie z.B. in `(++x) * (x--)`, sind unzulässig.
- Variablen sind Platzhalter für Werte (wie in der Mathematik) und keine Namen für veränderbare Speicherzellen.
- Funktionen haben keine Seiteneffekte.

Ohne diese Eigenschaften kann man leicht Programme konstruieren, bei denen die Reihenfolge der Auswertung arithmetischer Ausdrücke relevant ist.

Beispiel 5.3 (Relevanz der Auswertungsreihenfolge)**.**

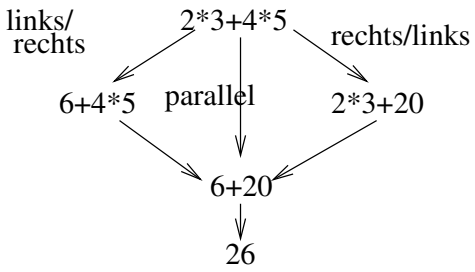
```
var y, z: int;

function f(x: int): int is
begin
   z := 2;
   return (2 * x)
end;

⋮
z := 1;
y := f(2) * z;
```

→ Bei Links-Rechts-Auswertung der Multiplikation: `y = 8`
→ Bei Rechts-Links-Auswertung der Multiplikation: `y = 4`

Die Vorteile der referenziellen Transparenz sind:

- Das mathematische Substitutionsprinzip gilt immer, d.h. man kann immer einen Ausdruck durch seinen Wert ersetzen, ohne dass sich am Ergebnis etwas ändert.
- Die operationale Semantik ist sehr einfach: „ersetze Gleiches durch Gleiches“.
- Das Prinzip ermöglicht einfache Optimierung, Transformation, und Verifikation von Programmen. Bei dem obigen Beispiel ist die Ersetzung von `f(2)` durch `4` im Programmcode eine unzulässige Optimierung.

- Das Prinzip ermöglicht eine flexible Auswertungsreihenfolge:



  Dies erlaubt eine einfache Ausnutzung paralleler Rechnerarchitekturen.

- Es entstehen lesbare, zuverlässige, weniger fehleranfällige Programme, da die Reihenfolgen von Zuweisungen nicht relevant sind.

Betrachten wir hierzu ein weiteres Beispiel.

Beispiel 5.4 (Sortieren mittels Quicksort)**.** Die klassische Formulierung in einer imperativen Sprache sieht etwa wie folgt aus:

```
procedure qsort(l, r: index);
var i,j: index; x, w: item
begin
   i := l; j := r;
   x := a[(l + r) div 2];
   repeat
      while a[i] < x do i := i + 1;
      while x < a[j] do j := j - 1;
      if i <= j then
      begin w := a[i]; a[i] := a[j]; a[j] := w;
         i := i + 1; j := j - 1
      end
   until i > j;
   if l < j then qsort(l, j);
   if i < r then qsort(i, r);
end
```

Hier kann man sehr viele Fehler machen, z. B. bei der Initialisierung, bei der Wahl der Abbruchbedingungen, der Reihenfolge der Zuweisung etc. Dies führt dazu, dass man es wohl kaum schafft, die erste Version des Programms fehlerfrei aufzuschreiben. Dagegen ist die Formulierung von Quicksort als funktionales Programm viel klarer und auch weniger fehleranfällig:

```
qsort []     = []
qsort (x:xs) = qsort (filter (< x) xs) ++ x : qsort (filter (>= x) xs)
```

Im Folgenden schauen wir uns die Syntax und Semantik funktionaler Programmiersprachen genauer an, um die Bedeutung dieses Programms genau zu verstehen. Als konkrete funktionale Programmiersprache betrachten wir die Sprache Haskell[1], die heutzutage als Standard für eine moderne rein funktionale Sprache gilt.

5.2 Syntax funktionaler Programmiersprachen

Ein **funktionales Programm** ist eine Menge von Funktionsdefinitionen (und Datentypdeklarationen, diese behandeln wir jedoch später). Eine **Funktionsdefinition** ist ähnlich wie in imperativen Sprachen, allerdings ist der Rumpf nur ein Ausdruck, da es keine Seiteneffekte gibt. Konkret erfolgt die Definition einer Funktion durch eine **Gleichung** oder **Regel**:

```
f x1 ... xn = e
```

Hier ist `f` der Funktionsname, $\texttt{x}_1,\ldots,\texttt{x}_n$ die Parameter der Funktion und `e` der Rumpf der Funktion.

Beispiel 5.5 (Funktionsdefinition)**.** Wir betrachten eine Funktion zum Quadrieren.

```
square x = x * x
```

Bei Funktionsdefinitionen ist Folgendes zu beachten:

- Die Anwendung von Funktionen auf Argumente erfolgt durch Hintereinanderschreibung, statt `f(x1,...,xn)` schreibt man `f x1 ... xn` oder auch `(f x1 ... xn)`.
- Die übliche Infixschreibweise bei (mathematischen) Operatoren ist zulässig, wie z. B. bei `3*4`.
- Das Ausrechnen einer Funktion entspricht der Ersetzung des Aufrufs durch den Rumpf, wobei die formalen Parameter durch die aktuellen Argumente ersetzt werden:

  ```
  square 5  → 5 * 5 → 25
  ```

- In Funktionsdefinitionen sind auch mehrere Alternativen mit Bedingungen möglich:

  ```
  f x1 ... xn | b1 = e1
              | b2 = e2
              ⋮
              | bn = en
  ```

[1] `http://www.haskell.org`

wobei `bn` auch `otherwise` sein kann, was für `True`, also immer erfüllt, steht.

Beispiel 5.6 (Bedingte Alternativen)**.**

```
fac n | n == 0 = 1
      | n >  0 = n * fac (n - 1)
```

Alternativ kann `fac` auch mit einer Fallunterscheidung definiert werden als

```
fac n = if n == 0 then 1 else n * fac (n - 1)
```

Eine deutlich besser lesbare Alternative zur Definition mittels bedingten Alternativen bieten **musterorientierte Definitionen**, die auf folgenden Ideen beruhen:

- Es gibt mehrere Gleichungen für eine Funktion.
- Man verwendet **Muster** (Datenterme) statt Variablen formale Parameter.
- Bedeutung musterorientierter Regeln: wende diese an, falls das Muster „passt“.

Der musterorientierten Stils hat viele Vorteile:

- kurze, prägnante Definitionen
- leicht lesbar
- leicht verifizierbar

Als Beispiel betrachten wir die Funktion `and` zur Konjunktion boolescher Werte:

- Musterorientiert:

  ```
  and True  x = x
  and False x = False
  ```

- mit Bedingungen:

  ```
  and x y | x == True  = y
          | x == False = False
  ```

- mit Fallunterscheidung:

  ```
  and x y = if x == True then y else False
  ```

5.2.1 Datentypen

In Haskell sind folgende elementare Typen wie üblich vordefiniert:

Wahrheitswerte: Typ `Bool`, Konstanten: `True`, `False`

Ganze Zahlen: Typ `Int` (oder `Integer` für beliebig große Zahlen), Konstanten: `0`, `1`, `-2`, ...

Gleitkommazahlen: Typ `Float` (oder auch `Double`), Konstanten: `0.0`, `1.5e-2`, ...

Zeichen: Typ `Char`, Konstanten: `'a'`, `'b'`, `'\n'`, ...

Zeichenketten: Typ `String` (Liste von `Char`), z. B. `"abcd"`

Eine Besonderheit funktionaler Sprachen ist die einfache Definition neuer **(algebraischer) Datentypen** durch Aufzählung der Konstruktoren (dies entspricht Vereinigungstypen), wobei auch eine Parametrisierung erlaubt ist. Die allgemeine Form einer Datentypdefinition ist:

`data` $T\ a_1\ \ldots\ a_n$ `=` $C_1\ \tau_{11}\ \ldots\ \tau_{1m_1}$ `|` $\ldots$ `|` $C_k\ \tau_{k1}\ \ldots\ \tau_{km_k}$

Hierbei sind

- T der Name des neuen Datentyps (Typname),
- $a_1, \ldots, a_n$ Typvariablen ($n \geq 0$), d. h. die Parameter des generischen Datentyps,
- $C_1, \ldots, C_k$ die Konstruktoren des Datentyps ($k > 0$), und
- $\tau_{i1}, \ldots, \tau_{im_i}$ die Argumenttypen des Konstruktors C_i ($m_i \geq 0$ für alle i), die aufgebaut sind aus anderen Typnamen und $a_1, \ldots, a_n$.

Mittels algebraischer Datentypen können wir verschiedene andere Typstrukturen definieren:

- Aufzählungstypen:

```
data Bool  = True | False
data Color = Red | Yellow | Blue | Green
```

- Parametrisierte Listen:

```
data List a = Nil | Cons a (List a)
```

  In Haskell sind diese mit einer bestimmten Syntax vordefiniert: Wir schreiben „`[`τ`]`“ statt „`List` τ“ und „`:`“ statt „`Cons`“, wobei „`:`“ als rechtassoziativer Infixoperator definiert ist. So können wir die Liste mit den Elementen 1,2,3 in Haskell wie folgt schreiben:

```
1:(2:(3:[]))   1:2:3:[]    [1,2,3]
```

- Binärbäume:

  ```
  data BTree a = Leaf a | Node (BTree a) (BTree a)
  ```

  So ist `Node (Leaf 1) (Node (Leaf 2) (Leaf 3)) :: BTree Int` ein möglicher Wert eines Binärbaumes über `Int`-Zahlen. Hierbei bedeutet „e `::` τ“, dass der Ausdruck e den Typ τ hat.

- Wir können auch allgemeine Bäume definieren, wobei die Knoten nun Listen von Teilbäumen besitzen können:

  ```
  data Tree a = Leaf a | Node [Tree a]
  ```

- Tupel sind vordefiniert:

  `(`e_1`, ..., `e_n`)` `::` `(`τ_1`, ..., `τ_n`)` ist ein Tupel, falls e_i `::` τ_i für $i = 1, \ldots, n$ gilt. So ist z. B. `(1, True, 'a') :: (Int, Bool, Char)` ein gültiges Tupel.

5.2.2 Allgemeine Funktionsdefinition

Allgemein können in Haskell Funktionen durch mehrere Gleichungen der Form

```
f t_1 ... t_n | b_1 = e_1
              | b_2 = e_2
              ⋮
              | b_n = e_n
```

definiert werden. Hierbei sind

- `f` der Funktionsname,
- $t_1, \ldots, t_n$ Muster, bestehend aus Konstruktoren und Variablen, wobei jede Variable höchstens einmal vorkommt,
- $b_1, \ldots, b_n$ boolesche Ausdrücke,
- $e_1, \ldots, e_n$ beliebige Ausdrücke, bestehend aus Variablen, Konstruktoren und Funktionsaufrufen, wobei die b_i und e_i nur Variablen aus $t_1, \ldots, t_n$ enthalten dürfen.

Beispiel 5.7 (Funktionsdefinition)**.** Konkatenation von Listen

```
conc []     ys = ys
conc (x:xs) ys = x : (conc xs ys)
```

Diese ist bereits vordefiniert in Haskell als Infixoperator „++“.

Beachte bei Funktionsdefinitionen Folgendes:

- Der musterorientierter Stil bedeutet keine Erhöhung der Berechnungsstärke, sondern sorgt für eine klarere Programmstruktur.
- Muster in linken Regelseiten entsprechen einer Kombination aus Prädikaten (hat das Argument diese Form?) und Selektoren (Zugriff auf Teilargumente). Somit vereinfachen Muster die Funktionsdefinitionen.

Beispiel 5.8 (Konkatenation ohne Muster)**.** Ohne Muster benötigen wir folgende Hilfsfunktionen als Selektoren:

- `head xs`: erstes Element von `xs`
- `tail xs`: Rest von `xs` (ohne `head xs`)

Nun können wir `conc` so definieren:

```
conc xs ys = if xs == []
               then ys
               else (head xs) : (conc (tail xs) ys)
```

Hier ist die Definition ohne Muster noch relativ einfach. Dies kann aber bei mehreren Regeln aufwändig werden:

```
or True  x     = True
or x     True  = True
or False False = False
```

müsste man ohne Muster schreiben als

```
or x y = if x == True
           then True
           else if y == True
                  then True
                  else if x == False && y == False
                         then False
                         else error "..."  -- unerreichbarer Fall
```

Eine weitere Übersetzungsalternative ist die Verwendung von `case`-Ausdrücken, diese werden intern in Haskell auch zur Umsetzung des musterorientierten Stils verwendet. `case`-Ausdrücke besitzen im Allgemeinen die folgende Form:

`case` e `of`
$C_1\ x_{11} \ldots x_{1r_1} \rightarrow e_1$

$\vdots$

$C_k \;\; x_{k1} \ldots x_{kr_k} \;\; \rightarrow e_k$

wobei e ein Ausdruck vom Typ τ, $C_1, \ldots, C_k$ Konstruktoren des gleichen Datentyps τ und x_{ij} Selektorvariablen sind, die in e_i benutzt werden können. Die Vorteile von `case`-Ausdrücken gegenüber `if-then-else` sind:

- implizite Selektoren durch die Selektorvariablen x_{ij}
- mehr als zwei Alternativen sind möglich

Beispiel 5.9 (`case`-Ausdrücke)**.**

```
conc xs ys = case xs of
               []   → ys
               z:zs → z : cons zs ys
```

Funktionale Programmiersprachen haben zudem einen „Pattern-Matching-Compiler“:

- Dieser dient der Übersetzung mehrerer Gleichungen für eine Funktion f in eine Gleichung der Form

  $$f \;\; x_1 \ldots x_n = e$$

  wobei in e auch `case`-Ausdrücke vorkommen können (vgl. `conc`).
- Die Strategie des Pattern Matching ist:
  - Teste für jede Regel die Muster von links nach rechts.
  - Probiere alle Regeln von oben nach unten aus.
  - Verwende die erste passende Regel.