Funktionale Programmierung
Sommersemester 2021

Prof. Dr. Jakob Rehof
TU Dortmund
LS XIV Software Engineering

Diese Vorlesung:

- DoberkatFP: Folien 260 – 298
  - Ein- und Ausgabe
- Wirkungen und Referentielle Transparenz

Was ist eine Funktion, mathematisch gesehen?

Eine (totale) Funktion $f : A \to B$ ist eine Relation $f \subseteq A \times B$

$$f = \{(x, y) \in A \times B \mid x \in A, y = f(x)\}$$

mit den Eigenschaften

- *Totalität*: $\forall x \in A.\ \exists y \in B.\ y = f(x)$
- *Funktionalität*: $\forall x_1, x_2 \in A.\ x_1 = x_2 \Rightarrow f(x_1) = f(x_2)$

Nicht zu verwechseln mit den weiteren Eigenschaften, die Funktionen haben können aber nicht haben müssen:

- *Injektivität*: $\forall x_1, x_2 \in A.\ f(x_1) = f(x_2) \Rightarrow x_1 = x_2$
- *Surjektivität*: $\forall y \in B.\ \exists x \in A.\ y = f(x)$

Übrigens, kennen Sie die **Kontraposition**?

$$(P \Rightarrow Q) \Leftrightarrow (\neg Q \Rightarrow \neg P)$$

Eine (totale) Funktion $f : A \to B$ ist eine Relation $f \subseteq A \times B$

$$f = \{(x, y) \in A \times B \mid x \in A, y = f(x)\}$$

mit den Eigenschaften

- *Totalität*: $\forall x \in A.\ \exists y \in B.\ y = f(x)$
- *Funktionalität*: $\forall x_1, x_2 \in A.\ f(x_1) \neq f(x_2) \Rightarrow x_1 \neq x_2$

Nicht zu verwechseln mit den weiteren Eigenschaften, die Funktionen haben können aber nicht haben müssen:

- *Injektivität*: $\forall x_1, x_2 \in A.\ x_1 \neq x_2 \Rightarrow f(x_1) \neq f(x_2)$
- *Surjektivität*: $\forall y \in B.\ \exists x \in A.\ y = f(x)$

Wirkung und Zustand

In einer *imperativen* Sprache kann ich schreiben:

```
int count := 0;

int f(int x) = {count := count + 1; return x + count}
```

wobei `count` eine globale Variable ist.
Das macht allerdings keine Funktion aus `f`! Denn ich kriege bei jedem Aufruf von `f` mit demselben Argument ein Unterschiedliches "Abbild":

```
y1 := f(5)
y2 := f(5)
```

Hier gilt mit Sicherheit `y1` $\neq$ `y2`. Also ist die Funktionalitätseigenschaft *nicht* erfüllt.

Dies liegt an der Verwendung von Operationen, die *Wirkungen* (*side effects*) auf einen globalen *Zustand* (hier durch `count` implementiert) haben.

Referentielle Transparenz

Eine Konsequenz der Funktionalitätseigenschaft ist die sogenannte **referentielle Transparenz**:

- *Ein Ausdruck ist mit seinem Wert in allen Kontexten austauschbar*

Daher, ob ich in einer *funktionalen* Sprache zum Beispiel schreibe

```
let y = f(5)
in
    (y, y)
```

oder ich schreibe

```
(f(5), f(5))
```

bleibt das Ergebnis identisch.

Store transformation

Man kann in einer funktionalen Sprache einen globalen Zustand simulieren, indem man *alle* Funktionen mit einem extra Argument ausstattet, welches den globalen Zustand repräsentiert.

Eine imperative "Funktion" $f : A \to B$ wird durch eine eigentliche Funktion $F : (A, Z) \to (B, Z)$ ersetzt, wobei Z der Typ des globalen Zustands ist. Den Übergang von f zu F nennt man manchmal *store transformation*. Man muss dafür sorgen, daß die "Wirkungen" von f auf den "Zustand" in F immer zurück gegeben wird und im Programm korrekt weiter gereicht wird (man nennt dies manchmal *store threading*).

Zum Beispiel:

```
F (x,count) = let count' = count+1 in (x+count',count')
```

mit Verwendung

```
let (y1,c1) = F (5,0)
    (y2,c2) = F (5,c1)
```

Die Funktion F ist eine richtige Funktion, mit referentieller Transparenz und allem.

So, what's the problem?

Das Problem bei *store transformation* ist allerdings, daß das *gesamte* Programm, mit dem totalen Programmkontext, transformiert werden muss. Der globale Zustand muss eben durch eine globale Transformation simuliert werden. Dies betrifft auch mögliche Bestandteile des Programms, die gar nicht vom Zustand abhängen!

The best of both worlds?

Könnten wir ein Konzept finden, mit welchem wir die Wirkungen auf den Zustand so *kappseln* könnten, dass nur die Bereiche des Programms, die wirklich vom Zustand abhängen, betroffen wären? Man spricht manchmal von *state encapsulation*.

Enter *monads*!

Stay tuned!

```
*Main> :t getLine
getLine :: IO String
*Main> x <- getLine
hello world!
*Main> :t x
x :: String
*Main> x
"hello world!"
*Main>
```

```
*Main> :t return
return :: Monad m => a -> m a
*Main> :t (return ())
(return ()) :: Monad m => m ()
*Main> :t (return "hello world!")
(return "hello world!") :: Monad m => m [Char]
*Main>
```

```
*Main> :i IO
type IO :: * -> *
newtype IO a
  = GHC.Types.IO (GHC.Prim.State# GHC.Prim.RealWorld
                  -> (# GHC.Prim.State# GHC.Prim.RealWorld, a #))
  -- Defined in ‘GHC.Types’
instance Applicative IO -- Defined in ‘GHC.Base’
instance Functor IO -- Defined in ‘GHC.Base’
instance Monad IO -- Defined in ‘GHC.Base’
instance Monoid a => Monoid (IO a) -- Defined in ‘GHC.Base’
instance Semigroup a => Semigroup (IO a) -- Defined in ‘GHC.Base’
instance MonadFail IO -- Defined in ‘Control.Monad.Fail’
*Main>
```

```
*Main>
*Main> :t (do x <- getLine; putStr x; return ())
(do x <- getLine; putStr x; return ()) :: IO ()
*Main>
```

```
*Main>
*Main> :t (do x <- getLine; putStr x; return ())
(do x <- getLine; putStr x; return ()) :: IO ()
*Main>
*Main>
*Main> y <- (do x <- getLine; putStr x; return ())
hellow world!
hellow world!*Main> :t y
y :: ()
*Main>
```

```
*Main> do handle <- openFile "outfile.hs" AppendMode;
hPutStr handle "again"; hClose handle; return ()

*Main> do handle <- openFile "outfile.hs" AppendMode;
hPutStr handle "again"; hClose handle; return ()

*Main>
```