Praktische funktionale Programmierung

Johannes Waldmann
Institut für Informatik
Universität Leipzig

Spiele

Programmierwettbewerb Connections (SS01)

`http://www.informatik.uni-leipzig.de/~joe/connections/`

- Interface (Java–Applet)
- Server (Haskell)
- Spielprogramme (C++, Java, Haskell, ...)

Spiel–Server

- Rezeption: Spieler empfangen
- Schiedsrichter:
  Spielzüge prüfen und verteilen
- Logger

Analog: Web–Server

- HTTP–Request empfangen
- Request bearbeiten
- Protokoll schreiben

Aktionen als Funktionen

Falsch:

```
writeFile :: FilePath -> String
getLine   :: String
```

Was ist

```
let xs = getLine in ( xs ++ xs ++ xs )
```

Nicht referentiell transparent!

Aktionen als Funktionen (II)

Richtig:

```
writeFile :: FilePath -> String ->
    Welt ->          Welt
getLine   ::
    Welt -> (String, Welt)

\ w0 -> let (xs0, w1) = getLine w0
            (xs1, w2) = getLine w1
            (xs2, w3) = getLine w2
        in  (xs0 ++ xs1 ++ xs2, w3)
```


Typ für Aktionen

```
type IO a =  Welt -> (a, Welt)
getLine   :: IO String
writeFile :: FilePath -> String -> IO ()
```

Die Welt nicht kopieren oder löschen!

Clean: man darf Type Welt benutzen, der Compiler prüft Unique–ness.

Haskell: nur IO a wird exportiert, und gewisse Kombinatoren.

Kombinieren von Aktionen

```
type IO a = Welt -> (a, Welt)

dann :: IO a -> IO b -> IO (a, b)
a `dann` b  =  \ w0 ->
    let (x1, w1) = a w0
        (x2, w2) = b w1
    in  ((x1, x2), w2)

getLine `dann` getLine
```


Elementare Kombinatoren

```
type IO a = Welt -> (a, Welt)

return :: a -> IO a
return x = \ w0 -> (x, w0)

(>>=) :: IO a -> (a -> IO b) -> IO b
a >>= f = \ w0 ->
    let (x1, w1) = a w0
        (b , w2) = f w1
        (x3, w3) = b w2
    in  (x3, w3)
```

Für Mathematiker: Die Struktur (`IO a`, `(>>=)`, **`return`**) ist eine Art Monoid.

Ein Haskell–Hauptprogramm

```
module Main (main) where
import System (getArgs)

main :: IO ()
main = getArgs >>= \ argv ->
    let xs = unwords argv
    in  putStrLn xs
```

Einfachere Notation:

```
main = do
    argv <- getArgs
    let xs = unwords argv
    putStrLn xs
```


Nebenläufigkeit

Threads innerhalb des Haskell–RTS verwaltet (ist für lazy evaluation sowieso nötig)

```
forkIO :: IO a -> IO ThreadId
killThread :: ThreadId -> IO ()
```

Der Main–Loop des Servers:

```
reception socket = do
    (h, host, port) <- accept socket
    forkIO ( handle h )
    reception socket
```


Kommunikation: Kanäle

```
data Chan a

newChan          :: IO (Chan a)
writeChan        :: Chan a -> a -> IO ()
readChan         :: Chan a -> IO a
```

implementiert mit Semaphoren

Merkmale, Vorteile

- scharfes Typsystem
- Trennung von Aktion und Wert
- Polymorphie, Funktionalität
  ⇒ modular und skalierbar
- threads sind lightweight

Zusammenfassung: *Haskell ist die beste imperative Programmiersprache der Welt.*

Weiteres

www.haskell.org

Skript *Praxis der Funktionalen Programmierung* (WS00)

http://www.informatik.uni-leipzig.de/~joe/edu/ws00/praxis/

Vorlesung *Kombinatorische Spieltheorie* (WS01)

http://www.informatik.uni-leipzig.de/~joe/edu/ws01/kst/

nächster Programmierwettbewerb (SS02)