

Fortgeschrittene Techniken der Funktionalen Programmierung
Vorlesung vom 08.12.09:
Nebenläufigkeit in Haskell: Abstraktionen und Ausnahmen

Christoph Lüth, Dennis Walter

Universität Bremen

Wintersemester 2009/10

1

Fahrplan

- ▶ Teil I: Monaden und fortgeschrittene Typen
- ▶ Teil II: Fortgeschrittene Datenstrukturen
- ▶ Teil III: Nebenläufigkeit
 - ▶ Grundlagen
 - ▶ Abstraktionen und Ausnahmebehandlung
 - ▶ Software Transactional Memory
- ▶ Teil IV: The Future of Programming

2

Tagesmenü

- ▶ Abstraktionen: Kanäle
- ▶ Fallbeispiel:
 - ▶ Talk
- ▶ Ausnahmebehandlung:
 - ▶ Erweiterbare Ausnahmen
 - ▶ Unscharfe Ausnahmen
 - ▶ Asynchrone Ausnahmen

3

Kanäle

- ▶ Typsicheres Lesen/Schreiben in FIFO-Ordnung
- ▶ Blockiert wenn leer

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

- ▶ Bonus: Duplizierbar ("Broadcast")

4

Kanäle

- ▶ Ein Kanal besteht aus Strom mit einem Lese- und Schreibende:

```
data Chan a = Chan (MVar (Stream a))
               (MVar (Stream a))
```

- ▶ Hier MVar, um Lesen/Schreiben zu synchronisieren
- ▶ Ein Strom ist MVar (ChItem a):
 - ▶ entweder leer,
 - ▶ oder enthält Werte aus Kopf a und Rest.

```
type Stream a = MVar (ChItem a)
data ChItem a = ChItem a (Stream a)
```

5

In einen Kanal schreiben

- ▶ Neues Ende (hole) anlegen
- ▶ Wert in altes Ende schreiben
- ▶ Zeiger auf neues Ende setzen

```
writeChan :: Chan a -> a -> IO ()
writeChan (Chan _ write) val = do
  new_hole <- newEmptyMVar
  old_hole <- takeMVar write
  putMVar old_hole (ChItem val new_hole)
  putMVar write new_hole
```

- ▶ Kann nicht blockieren — write immer gefüllt.
- ▶ Original-Code benutzt modifyMVar — Ausnahmesicher!

6

Aus Kanal lesen

- ▶ Anfang auslesen, Anfangszeiger weitersetzen
- ▶ Kann blockieren (*) wenn Kanal leer

```
readChan :: Chan a -> IO a
readChan (Chan read _) = do
  read_end <- takeMVar read
  (ChItem val new_read_end) <- readMVar read_end —*
  putMVar read new_read_end
  return val
```

- ▶ readMVar :: MVar a -> IO a liest MVar, schreibt Wert zurück.
- ▶ readMVar statt takeMVar, um Duplikation zu ermöglichen

7

Neuen Kanal erzeugen

- ▶ Lese-Ende = Schreib-Ende

```
newChan :: IO (Chan a)
newChan = do
  hole <- newEmptyMVar
  read <- newMVar hole
  write <- newMVar hole
  return (Chan read write)
```

8

Weitere Kanalfunktionen

- ▶ Zeichen wieder vorne einhängen:

```
unGetChan :: Chan a → a → IO ()
```

- ▶ Kanal duplizieren (Broadcast):

```
dupChan :: Chan a → IO (Chan a)
```

- ▶ Kanalinhalt als (unendliche) Liste:

```
getChanContents :: Chan a → IO [a]
```

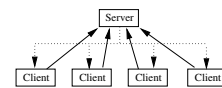
- ▶ Auswertung terminiert nicht, sondern blockiert

9

Fallbeispiel: Talk

- ▶ Ziel: ein Programm, um sich über das Internet zu unterhalten (talk, IRC, etc.)

- ▶ Verteilte Architektur:



- ▶ Hier: Implementierung des Servers

- ▶ Netzverbindungen durch **Socket**

10

Socketprogrammierung

- ▶ Socket erzeugen, an Namen binden, mit `listen` Verbindungsbereitschaft anzeigen

- ▶ Zustandsbasierte Verbindung:

- ▶ Serverseite: mit `accept` auf eingehende Verbindungen warten
- ▶ Jede Verbindung erzeugt neuen Filedescriptor
⇒ inhärent nebenläufiges Problem!
- ▶ Clientseite: Mit `connect` Verbindung aufnehmen.

- ▶ Zustandslose Verbindung: `sendTo` zum Senden, `recvFrom` zum Empfangen.

- ▶ GHC-Modul `Network`

- ▶ Low-level Funktionen in `Network.Socket`

11

Das Modul Network

- ▶ Sockets:

```
type Socket
data PortID = Service String — z.B. "ftp"
              | PortNumber PortNumber
              | UnixSocket String — Socket mit Namen
type Hostname = String
instance Num PortNumber
```

- ▶ Zustandsbasiert:

```
listenOn :: PortID → IO Socket
accept :: Socket → IO (Handle, Hostname, PortNumber)
connectTo :: Hostname → PortID → IO Handle
```

- ▶ Zustandslos:

```
sendTo :: Hostname → PortID → String → IO ()
recvFrom :: Hostname → PortID → IO String
```

12

Serverarchitektur

- ▶ Ein Kanal zur Nachrichtenverbreitung:

- ▶ eine Nachricht, viele Empfänger (**broadcast**)
- ▶ Realisierung mittels `dupChan`

- ▶ Zentraler Scheduler

- ▶ Für jede ankommende Verbindung neuer Thread:

- ▶ Nachrichten vom Socket auf den Kanal schreiben
- ▶ Nachrichten vom Kanal in den Socket schreiben

- ▶ **Problem:** Wie aus Socket **oder** Kanal lesen wenn beide blockieren?

- ▶ **Lösung:** Zwei Threads

- ▶ Client: `telnet`

13

Talk 0.1: Hauptprogramm

```
main = do
  a:_ ← getArgs
  let p = fromInteger (read a)
  s ← listenOn (PortNumber p)
  ch ← newChan
  loop s ch
```

14

Talk 0.1: Hauptschleife

```
loop s ch = forever $ do
  (handle, wh, p) ← accept s
  hSetBuffering handle NoBuffering
  putStrLn $ "New connection from " ++ wh ++
    " on port " ++ show p
  ch2 ← dupChan ch
  forkIO (newUser handle ch2)
```

15

Talk 0.1: Benutzerprozess

```
newUser :: Handle → Chan String → IO ()
newUser socket msgch =
  forkIO (forever read) >> forever write where
    read :: IO ()
    read = hGetLine socket >>= writeChan msgch
    write :: IO ()
    write = readChan msgch >>= hPutStrLn socket
```

16

Talk 0.1: Zusammenfassung

Nachteile:

- ▶ Nachrichten stauen sich im Kanal
- ▶ Keine Fehlerbehandlung
- ▶ Benutzer anonym

17

Ausnahmebehandlung in Haskell98

- ▶ Haskell 98: Fehler leben im IO-Monaden.

- ▶ Fehler fangen:

```
catch :: IO a → (IOError → IO a) → IO a
```

- ▶ Variante: try :: IO a → IO (Either IOError a)

- ▶ Fehler erzeugen:

```
userError :: String → IOError  
ioError :: IOError → IO a
```

- ▶ Oder durch andere Operationen im IO-Monaden.

18

Fehler analysieren

- ▶ Funktionen, die im Handler benutzt werden können:

```
isAlreadyExistsError :: IOError → Bool  
isDoesNotExistError :: IOError → Bool  
isAlreadyInUseError :: IOError → Bool  
isFullError :: IOError → Bool  
isEOFError :: IOError → Bool  
isIllegalOperation :: IOError → Bool  
isPermissionError :: IOError → Bool  
isUserError :: IOError → Bool  
  
ioeGetErrorString :: IOError → String  
ioeGetHandle :: IOError → Maybe Handle  
ioeGetFileName :: IOError → Maybe FilePath
```

19

Talk 0.2: Hauptschleife

```
loop s ch = forever $ do  
  (handle, wh, p) ← accept s  
  hSetBuffering handle NoBuffering  
  installHandler sigPIPE Ignore Nothing  
  putStrLn $ "New connection from " ++ wh ++  
    " on port " ++ show p  
  ch2 ← dupChan ch  
  forkIO (catch (newUser handle wh ch2)  
    (λ_ → hClose handle))
```

- ▶ Fehlerbehandlung für newUser (kein guter Stil)
- ▶ SIGPIPE ignorieren

20

Talk 0.2: Benutzerprozess

Teil 1: Anmeldeprozedur

```
newUser s wh msgch = do  
  hPutStrLn s "Hello there. Please send your nickname."  
  nick ← do nm ← hGetLine s  
    return (filter (not . isControl) nm)  
  hPutStrLn s $ "Nice to meet you, " ++ nick ++ "!"  
  writeChan msgch $ nick ++ "@" ++ wh ++ "_has_joined."
```

(Fortsetzung)

21

Talk 0.2: Benutzerprozess

Teil 2: Hauptschleife:

```
wp ← forkIO write  
catch (read ((nick ++ ": ") ++)) $ λe → do  
  killThread wp  
  writeChan msgch $  
    if isEOFError e then nick ++ "@" ++ wh ++ "_has_left."  
    else nick ++ "@" ++ wh ++ "_left hastily (" ++  
      ioeGetErrorString e ++ ")"  
  hClose s where  
    read :: (String → String) → IO ()  
    read f = forever $  
      hGetLine s >>= writeChan msgch . f  
    write :: IO ()  
    write = forever $ readChan msgch >>= hPutStrLn s
```

22

Talk 0.2: Zusammenfassung

Vorteile:

- ▶ Robust
- ▶ Fehlerbehandlung für Benutzerprozess
- ▶ Anmeldeprozedur: Benutzer hat Namen
- ▶ Schnell verkaufen!

23

Probleme mit der Ausnahmebehandlung in Haskell98

- ▶ Keine Ausnahmebehandlung für rein funktionalen Code.
 - ▶ error :: String → a bricht Programmausführung ab;
 - ▶ z.B. Fehler bei read :: Read a ⇒ String → a?
 - ▶ readIO :: Read a ⇒ String → IO a wirft Ausnahme
 - ▶ Laufzeitfehler (pattern match, fehlende Klassenmethoden, ...)
- ▶ Keine Behandlung von asynchronen Ausnahmen möglich.
 - ▶ Nebenläufige Fehler, e.g. stack overflow, Speichermangel, Interrupts;

24

Probleme mit rein funktionalen Ausnahmen.

- ▶ Warum nicht einfach `throw :: Exception → a`?
- ▶ Wird die Ausnahme geworfen?

```
length [throw exception]
```
- ▶ Abhängig von Tiefe der Auswertung (wertet `length` Argument aus?)
- ▶ Welche Ausnahme wird geworfen:

```
throw ex1 + throw ex2
```
- ▶ Abhängig von Reihenfolge der Auswertung der Argumente
- ▶ Aber: Auswertungsreihenfolge in Haskell98 **unspezifiziert!**

25

Unschärfe Ausnahmen.

- ▶ Normale Ausnahmen: Wert eines Ausdrucks = Normaler Wert oder Ausnahme

```
data Maybe a = Just a | Nothing
data Either a = Left String | Right a
```
- ▶ Unschärfe Ausnahmen: Wert eines Ausdrucks = Normaler Wert oder Menge von möglichen Ausnahmen
 - ▶ Menge wird nicht konstruiert — semantisches Konstrukt.

26

Unschärfe Ausnahmen fangen.

- ▶ Ausnahmen fangen ist monadisch:
 - ▶ Funktion `bogus :: a → (Exception → a) → a` hätte alten Probleme
- ▶ Determinisierung trennen von Ausnahmebehandlung:
 - ▶ `evaluate :: a → IO a` wertet Ausdruck aus, wirft ggf. mögliche Ausnahme.
 - ▶ Ausnahme durch Auswertungsreihenfolge bestimmt.
- ▶ `catch :: IO a → (Exception → IO a) → IO a` wie vorher.
- ▶ Unschärfe Ausnahmen können **überall** geworfen, aber nur im IO-Monaden gefangen werden.

27

Asynchrone Ausnahmen

- ▶ Modelliert durch

```
throwTo :: ThreadId → Exception → IO ()
```
- ▶ Ausnahme wird in **anderem** Thread geworfen.
- ▶ Modelliert **alle Situationen** wie Interrupts etc.

28

Asynchrone Ausnahmen: Beispiel

- ▶ Parallele Auswertung zweier IO-Statements:
- ▶ Wer zuerst fertig ist beendet Auswertung.

```
parIO :: IO a → IO a → IO a
parIO a1 a2 =
  do m ← newEmptyMVar
     c1 ← forkIO (a1 >>= putMVar m)
     c2 ← forkIO (a2 >>= putMVar m)
     r ← takeMVar m
     throwTo c1 Kill
     throwTo c2 Kill
     return r
```

29

Asynchrone Ausnahmen: Beispiel

- ▶ Timeout-Operator:
- ▶ Wenn kein Ergebnis nach n Mikrosekunden, Nothing

```
timeout :: Int → IO a → IO (Maybe a)
timeout n a = parIO (r ← a; return (Just r))
                  (threadDelay n; return Nothing)
```

30

Unschärfe Ausnahmen: Benutzung

- ▶ Zur Benutzung: `import Control.Exception` (nur `ghc`)
- ▶ Um **Erweiterbarkeit** zu gewährleisten:
 - ▶ Typklasse `Exception`, alle Ausnahmen sind Instanzen
 - ▶ Achtung, erst seit `ghc 6.10`.
- ▶ **Achtung:** per default **normale** Ausnahmen (Haskell98) definiert
 - ▶ Überlagerung durch `Import` oder Disambiguierung
- ▶ Ausnahmen fangen:

```
catch :: Exception e ⇒ IO a → (e → IO a) → IO a
try :: Exception e ⇒ IO a → IO (Either e a)
```

31

Vorsicht bei Ausnahmen

- ▶ Ausnahmen und Nebenläufigkeit
- ▶ Ausnahmen können in anderen Thread geworfen werden!

```
ch ← newChan
forkIO (forever $ do readChan ch >>= putStrLn)
catch (do let x = ...
          writeChan ch x)
      (λe → ...)
```

- ▶ Ausnahme wird in reader-Thread geworfen!
 - ▶ Abhilfe: Auswertung mit `evaluate` forcieren.

32

Ausnahmen: Achtung!

- ▶ Fehlerabfrage ersetzt **keine** Ausnahmebehandlung:

```
b ← doesDirectoryExist name  
when (not b) $ createDirectory name
```

- ▶ Zweite Aktion **kann** fehlschlagen!

33

Zusammenfassung

- ▶ **Kanäle**: Nützliche Kommunikationsabstraktion
- ▶ **Unschärfe** Ausnahmen:
 - ▶ Können in beliebigem Code auftreten
 - ▶ Werden im IO-Monaden gefangen
- ▶ **Asynchrone** Ausnahmen:
 - ▶ Werden in anderem Thread ausgelöst
- ▶ Ausnahmebehandlung:
 - ▶ **Essentiell** für robuste Programmierung
 - ▶ **Nur** Ausnahmen fangen, die man behandelt!
 - ▶ Fehlerabfrage ersetzt **keine** Ausnahmebehandlung

34