Fortgeschrittene Funktionale Programmierung

Teil 7: Ausnahmen & Nebenläufigkeit

Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

15. Januar 2019

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Teil 7: Ausnahmen & Nebenläufigkeit

Ausnahmen

Manchmal können Berechnungen fehlschlagen oder sind undurchführbar.

Wir haben bereits gesehen, wie wir solche Fälle mithilfe der Typen `Maybe a` und `Either a b` behandeln können:

```
safeHead :: [a] -> Maybe a
safeHead   []  = Nothing
safeHead (h:_) = Just h
```

Dies ist auch prinzipiell die beste Methode, um mit Ausnahmen umzugehen: Man erkennt direkt am Typ einer Funktion, das Ausnahmen eintreten können!

Nachteile: Behandlung einer Ausnahme kann manchmal nur an ganz anderer Stelle im Programm durchgeführt werden.
Plötzlich taucht überall ein `Maybe` auf, was lästig sein kein.

Ausnahmen mit Monaden

Lösung: `Maybe` und `Either a` kann man als Monaden auffassen, wodurch das Herumreichen eines Fehlers implizit wird.

Mit einer speziell zugeschnittenen Monade kann man so auch verschiedene mögliche Ausnahmen beschreiben und behandeln:

```
data Ausnahme = Bauchschmerzen | Kopfweh | Zahnschmerzen Int
type Gesundheit a = Either Ausnahme a   --besser newtype verwenden
-- instance Monad Gesundheit            --hier automatisch über Either

catch :: Gesundheit a -> (Ausnahme -> Gesundheit a) -> Gesundheit a
catch tat abhilfe = case tat of
  (Left ausnahme) -> abhilfe ausnahme
  anderes         -> anderes

hilfe Kopfweh = Right nimmAspirin -- Fehlerbehandlung
hilfe anderes = Left  anderes -- andere Leiden nicht abgefangen
```

Nachteile: Die Monade kann machmal Laziness verringern.

Ausnahmen in GHC

Wenn wir nach diesem Schema verfahren, dann kennzeichnet die Monade also alle möglichen Ausnahmen von vornherein!

Dennoch gibt es in Haskell auch noch implizit auftretende Ausnahmen, welche wir leider nicht am Typ erkennen können:

```
> head []
*** Exception: Prelude.head: empty list
> undefined
*** Exception: Prelude.undefined
> error "Pfui Deife!"
*** Exception: Pfui Deife!
```

Insbesondere der Typ der `error` Funktion zeigt dies gar nicht an:

```
> :t error
error :: String -> a
```

Ausnahmen in GHC

Solche Ausnahmen sind durch die Klasse `Exception` aus dem Modul `Control.Exception` erfasst:

```
class (Typeable e, Show e) => Exception e where ...
    toException   :: Exception e -> SomeException Source
    fromException :: SomeException -> Maybe e
```

Die eingebaute Funktion `throw` erlaubt es, eine Ausnahme mit einer beliebigen Instanz der Typklasse `Exception` an einer beliebigen Stelle im Code zu werfen:

```
throw   :: Exception e => e ->    a -- abhängig Lazy-Ev.
throwIO :: Exception e => e -> IO a
```

Alle Instanzen der Typklasse `Exception` bilden eine erweiterbare Hierarchie, an dessen Wurzel der Typ `SomeException` steht.

verwendet Typen höheren Ranges und die spezielle Typklasse `Typeable` zur Typ Reifikation

Eigene Ausnahmen definieren

```
import Data.Typeable
import Control.Exception

data Ausnahme = Bauchschmerzen | Kopfweh
              | Zahnschmerzen Int
  deriving (Typeable, Show)

instance Exception Ausnahme
  -- Defaults reichen aus, also kein 'where'
```

Diese Deklarationen erweitern die Hierarchie der Ausnahmen mit Datentyp zur Beschreibung einer speziellen Art von Ausnahmen.

```
> throw (Zahnschmerzen 3)
*** Exception: Zahnschmerzen 3
```

Ausnahme Hierarchie

Auch Standarbibliothek definiert Ausnahmen nach diesem Muster:

```
newtype ErrorCall = ErrorCall String
    deriving (Typeable)

instance Show ErrorCall where
  showsPrec _ (ErrorCall err) = showString err

instance Exception ErrorCall -- default

error :: String -> a
error s = throw (ErrorCall s)
```

Ausnutzen können wir die Hierarchie bei der Ausnahmebehandlung.

Ausnahmebehandlung

Ausnahmen können nur innerhalb der alles umfassenden IO-Monade behandelt werden:

```
module Control.Exception where
  catch  :: Exception e => IO a -> (e -> IO a) -> IO a
  handle :: Exception e => (e -> IO a) -> IO a -> IO a
  handle = flip catch
```

Der Typ legt fest, welche Art von Exceptions gefangen werden:

```
> catch (throw Kopfweh) (\e -> print (e :: Ausnahme))
Kopfweh
> catch (throw $ ErrorCall "Bad") (\e -> print (e :: Ausnahme))
*** Exception: Bad
> catch (throw $ ErrorCall "Bad") (\e -> print (e :: SomeException))
Bad
> catch (throw Kopfweh) (\e -> print (e :: SomeException))
Kopfweh
```

Typ-Angabe meist besser per Typsignatur realisieren

Fehler und Ausnahmen

Hinweis: Es ist nicht ratsam, blind alle möglichen Ausnahmen zu fangen, wie hier in diesem Beispiel:

```
handle (\e -> print (e :: SomeException)) (...)
```

Nur Ausnahmen fangen, welche man sinnvoll behandeln kann!

Jede ungefangene **Ausnahme**/**Exception** ist ein **Fehler**/**Error**.
Ein Fehler führt immer zum Abbruch des Programms.

- **Exception** Ausnahme, welche speziell behandelt werden muss
- **Error** Programmierfehler, d.h. Programmierer hat nicht alle Fälle korrekt durchdacht, auch Endlosschleifen, etc.

Beispiel: Datei soll geöffnet werden, doch die Datei existiert nicht. Kann durch Programmierer abgefangen worden sein ($\Rightarrow$ Exception) oder wurde (un-)bewusst vergessen ($\Rightarrow$ Error).

Die genauere Unterscheidung ist diskutabel und unterscheidet sich auch in verschiedenen Programmiersprachen.

Ausnahmebehandlung II

```
catch  :: Exception e => IO a -> (e -> IO a) -> IO a
-- Zur regulären Ausnahmebehandlung:
try :: Exception e => IO a -> IO (Either e a)
-- Zum Aufräumen im Fehlerfall:
onException :: IO a -> IO b -> IO a
finally     :: IO a -> IO b -> IO a
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
```

- `try` eignet zur gewöhnliche Ausnahmebehandlung besser als `catch`, da Ausnahme zum greifbaren Datum wird
- `onException` führt bei Ausnahme noch Aufräumaktion aus
- `finally` führt die Aufräumaktion immer aus
- `bracket open close work` führt `close` immer aus

`onException`, `finally`, `bracket open close work` reichen eine eventuelle Ausnahme immer nach außen weiter $\Rightarrow$ Fehler

Zusammenfassung Exceptions

- Ausnahmen können überall geworfen werden
- Ausnahmen können nur in IO-Monade abgefangen werden
- Ausnahme-Typen bilden Hierarchie, welche um benutzerdefinierbare Ausnahmen erweitert werden können
- Ausnahmen nur gezielt abfangen um
  - konkrete Probleme zu beheben
  - notwendige Aufräumaktionen durchzuführen
- Ausnahme-Behandlung nur sparsam einsetzen, da dies schnell zu undurchsichtigem Code führt;
  besser `Maybe` oder `Either` oder ähnliche Typen verwenden

Grundlagen

Paralleles Rechnen: Ziel ist *schnelle* Ausführung von Programmen durch gleichzeitige Verwendung mehrerer Prozessoren.

Dagegen bedeutet **Nebenläufigkeit** engl. Concurrency
nicht-deterministische Berechnungen durch zufällig abwechselnd ausgeführte interagierende Prozesse,
z.B. Reaktion auf gleichzeitige externe Ereignisse.

Dazu reicht auch ein einzelner Prozessorkern aus, auf dem alle nebenläufigen abwechselnd Threads ausgeführt werden

mehrere Kerne können natürlich direkt genutzt werden

Beispiele:

- Auf Mausklicks reagieren, während Video abgespielt wird
- Webserver beantwortet Anfragen mehrerer Benutzer simultan

$\Rightarrow$ Nebenläufigkeit strukturiert ein Programm in mehrere *asynchron* ablaufende Einheiten!

Probleme bei Nebenläufigkeit

Bei nebenläufigen Berechnungen mit mehreren Threads können u.a. folgende Probleme auftreten:

Race-Condition Verschiedene Threads können sich gegenseitig beeinflussen. Da manchmal ein Thread schneller als ein anderer abgehandelt wird, und die Möglichkeiten der Verzahnung immens sind, ist das Gesamtergebnis der Berechnung nicht vorhersagbar.

Deadlock Ein Thread wartet auf das Ergebnis eines anderen Threads, welche direkt oder indirekt selbst auf das Ergebnis des ersten Threads wartet. Die Berechnung kommt somit zum erliegen.

Im Gegensatz zur rein funktionalen *parallelen* Berechnung lassen sich diese Probleme bei *nebenläufigen* Programmen *nicht* generell im Vorhinein vermeiden!

Concurrent Haskell

Explizite Parallelität bietet das Modul `Control.Concurrent` mit Bibliotheksfunktionen zur Erzeugung und Kontrolle von nebenläufigen Berechnungen, sogenannten **IO-Threads**.

Ein IO-Thread wird nebenläufig zum Hauptthread abgearbeitet. Auch wenn nur ein Prozessorkern verfügbar ist, soll die Illusion entstehen, als würden beide Threads gleichzeitig agieren.

IO-Threads sind explizite Objekte, die im Code kontrolliert werden.

```
myThreadId :: IO ThreadId
forkIO     :: IO () -> IO ThreadId
forkOS     :: IO () -> IO ThreadId
```

`forkIO` erzeugt Thread, identifiziert über `ThreadId`

`forkOS` erzeugt echten OS-Thread; teurer aber manchmal notwendig bei Verwendung sprach-fremder Bibliotheken über FFI.

Beispiel: Nebenläufigkeit

```
import Control.Concurrent
import Control.Monad
import System.IO

main = do hSetBuffering stdout NoBuffering
          forkIO (replicateM_ 2000 (putChar 'A'))
          replicateM_ 2000 (putChar 'B')
```

Im Beispiel wird zuerst der Ausgabepuffer abgeschaltet, damit man genau sieht, welcher Thread wann agiert. Der eine Thread schreibt 2000 mal `A`, der andere völlig unabhängig davon 2000 mal `B`.
Die Verzahnung ist zufällig. GHC bemüht sich jedoch, alle Threads gleichmässig auszführen: $\Rightarrow$ Fair Scheduling

```
BBBBABABABABABABABABABABABABABABABABABAABABABABA...
```

Fehlende Synchronisation

```
import Control.Concurrent

fib :: Integer -> Integer
fib n | n < 2      = 1
      | otherwise = fib (n-1) + fib (n-2)

showFib s n = putStrLn $ s ++ (show $ fib n)

main = do putStrLn "Creating Threads."
          forkIO $ showFib "A" 42
          forkIO $ showFib "B" 38
          forkIO $ showFib "C" 40
          putStrLn "Done."
```

Das Programm wird beendet, sobald der Hauptthread beendet wird.

$\Rightarrow$ Notwendigkeit zur Synchronisation zwischen Threads

Fehlende Synchronisation

```
import Control.Concurrent

fib :: Integer -> Integer
fib n | n < 2      = 1
      | otherwise = fib (n-1) + fib (n-2)

showFib s n = putStrLn $ s ++ (show $ fib n)

main = do putStrLn "Creating Threads."
          forkIO $ showFib "A" 42
          forkIO $ showFib "B" 38
          showFib "C" 40   -- auch nicht besser
          putStrLn "Done."
```

Das Programm wird beendet, sobald der Hauptthread beendet wird.

$\Rightarrow$ Notwendigkeit zur Synchronisation zwischen Threads

Implizite Synchronisation

```
import Control.Concurrent

fib n | n < 2      = 1
      | otherwise = fib (n-1) + fib (n-2)
x = fib 42        -- top-level definition (lokal geht auch)

main = do putStrLn "Creating Threads."
          forkIO $ print ("A",x)
          forkIO $ print ("B",x)
          print ("C",x)
          putStrLn "Done."
```

Greifen mehrere Threads auf den gleichen Thunk zu, findet eine **implizite Synchronisation** statt: Die späteren Threads warten, bis die Berechnung des Thunks durch den zuerst zugreifenden Thread beendet ist. Dies für unabhängige Threads meist unerwünscht.

Explizite Synchronisation: MVars

Kommunikation zwischen IO-Threads mit *synchronisierten* shared-memory Variablen **MVar** aus Modul `Control.Concurrent.MVar`

- `MVar a` ist eine veränderliche Speicherstelle des Typs `a`
- Zugriff innerhalb IO-Monade, sonst keine Einschränkungen
- MVar kann leer oder besetzt sein

Drei grundlegenden Operationen:

```
newEmptyMVar :: IO (MVar a)
takeMVar     :: MVar a ->      IO a
putMVar      :: MVar a -> a -> IO ()
```

Semantik:

	MVar leer	MVar besetzt
takeMVar	Blockiert	liefert & leert MVar
putMVar	Setzt MVar	Blockiert

Beispiel für MVars: Rendezvous

```
task :: MVar Int -> MVar Int -> IO ()
task receive send = do        -- initiale Berechnung hier
  v <- takeMVar receive       -- warten auf receive
  let res = 3 * v             -- Berechnung des Tasks
  putMVar send res            -- send besetzen

main = do
  aMVar <- newEmptyMVar
  bMVar <- newEmptyMVar
  cMVar <- newEmptyMVar
  putMVar aMVar 1
  forkIO $ task aMVar bMVar -- wartet auf a
  forkIO $ task bMVar cMVar -- wartet auf b
  v <- takeMVar cMVar         -- warten auf c
  print v
```

Drei Threads laufen unabhängig nebenläufig ab; bei `takeMVar` wird jedoch gewartet bis dort ein Wert bereit steht.

Beispiel für Deadlock

```
task :: MVar Int -> MVar Int -> IO ()
task receive send = do        -- initiale Berechnung hier
  v <- takeMVar receive       -- warten auf receive
  let res = 3 * v             -- Berechnung des Tasks
  putMVar send res            -- send besetzen

main = do
  aMVar <- newMVar 1
  bMVar <- newEmptyMVar
  cMVar <- newEmptyMVar
  forkIO $ task cMVar bMVar
  forkIO $ task bMVar cMVar
  v <- takeMVar cMVar
  print v
```

Alle Threads sind durch `takeMVar` blockiert;
kein Thread kann `putMVar` ausführen:

```
> ./Rendezvous01
Rendezvous01: thread blocked indefinitely in an MVar operation
```

Asychnrones Mischen

```
task :: MVar Integer -> Integer -> IO ()
task out n = putMVar out $ fib n

main = do putStrLn "Computing..."
          x <- newEmptyMVar
          forkIO $ task x 42
          forkIO $ task x 38
          v <- takeMVar x
          print v
          v <- takeMVar x
          print v
          putStrLn "Done."
```

MVars dürfen mehrfach gelesen und beschrieben werden

im Gegensatz zu IVars

```
Computing...
433494437
63245986
Done.
```

MVars sind faul

```
task :: MVar Integer -> Integer -> IO ()
task out n = do putStrLn $ "Start " ++ (show n)
                putMVar out $ fib n
                putStrLn $ "Ende " ++ (show n)

main =       do putStrLn "Computing..."
                x <- newEmptyMVar
                forkIO $ task x 42
                forkIO $ task x 38
                v <- takeMVar x
                print v
                putStrLn "Done."
```

MVars sind faul

```
task :: MVar Integer -> Integer -> IO ()
task out n = do putStrLn $ "Start " ++ (show n)
                putMVar out $ fib n
                putStrLn $ "Ende " ++ (show n)

main =       do putStrLn "Computing..."
                x <- newEmptyMVar
                forkIO $ task x 42
                forkIO $ task x 38
                v <- takeMVar x
                print v
                putStrLn "Done."
```

MVar erzwingt keine Auswertung im Gegensatz zu IVar:

```
Computing...
Start 44
Start 38
Ende 44
Ende 38
1134903170
Done.
```

MVars sind faul

```
task :: MVar Integer -> Integer -> IO ()
task out n = do putStrLn $ "Start " ++ (show n)
                putMVar out $! fib n -- Auswertung erzwingen!
                putStrLn $ "Ende " ++ (show n)

main =       do putStrLn "Computing..."
                x <- newEmptyMVar
                forkIO $ task x 42
                forkIO $ task x 38
                v <- takeMVar x
                print v
                putStrLn "Done."
```

MVar erzwingt keine Auswertung im Gegensatz zu IVar:

```
Computing...
Start 44
Start 38
Ende 44
Ende 38
1134903170
Done.
```

Explizite Synchronisation: MVars

MVars erlauben direkt

- gemeinsame Verwendung von Datenstrukturen, z.B. write-locks für Dateien
- sind Kommunikationskanäle ohne Puffer

Da MVars für beliebige Typen deklariert werden können, ist es relativ leicht möglich, Kommunikationskanäle mit unbegrenzten Puffer (FIFO Warteschlangen) zu erstellen.

Diese gibt es aber auch schon fertig: Control.Concurrent.Chan

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

Schreiben blockiert nie. Ist der Channel leer, so blockiert ein Leserversuch bis der Channel wieder gefüllt wird.

Übersicht MVar Operationen

Verfügbare Operationen aus Modul `Control.Concurrent.MVar`:

Blocking

`takeMVar :: MVar a -> IO a`	nimmt nicht-leere MVar
`putMVar :: MVar a -> a -> IO ()`	schreibt leere MVar
`readMVar :: MVar a -> IO a`	liest nicht-leere MVar
`swapMVar :: MVar a -> a -> IO a`	tauscht MVar
`modifyMVar_ :: MVar a -> (a -> IO a) -> IO ()`	Fkt.anw.

Non-blocking

`newMVar :: a -> IO (MVar a)`	erzeugte initialisierte MVar
`newEmptyMVar :: IO (MVar a)`	erzeugte neue leere MVar
`isEmptyMVar :: MVar a -> IO Bool`	testet ob MVar leer
`tryTakeMVar :: MVar a -> IO (Maybe a)`	
`tryPutMVar :: MVar a -> a -> IO Bool`	

Fallstrick bei MVar

Eine MVar des Typs `MVar (IO a)` erzielt vermutlich selten das gewünschte Verhalten, wie z.B. hier:

```
mv <- newEmptyMVar :: MVar (IO ())
forkIO $ putMVar mv $ putStrLn "Test"
```

Hier wird eine IO-Aktion in einer MVar abgespeichert, aber eben nicht ausgeführt! Der Seiteneffekt Textausgabe tritt nicht auf.

```
mv <- newEmptyMVar :: MVar ()
forkIO $ do { putStrLn "Test";  putMVar mv () }
```

Hier wird zuerst eine IO-Aktion ausgeführt, und nach deren Beendigung wird die MVar gesetzt, um das Ende der nebenläufigen Berechnung zu signalisieren.

Erinnerung: Monadische Hintereinanderausführung mit `(>>=)` grantiert lediglich, dass Seiteneffekte Reihenfolge einhalten:

```
do { these_effects_first; afterwards_those_effects }
```

Zusammenfassung MVar

- Eine MVar kann mehrfach beschrieben werden, aber immer nur, wenn diese leer ist. Ansonsten blockiert ein schreibender Thread so lange, bis ein anderer Thread die MVar geleert hat.
- Eine MVar kann mehrfach gelesen werden, aber immer nur, wenn diese besetzt ist. Ansonsten blockiert ein lesender Thread so lange, bis ein anderer Thread die MVar gesetzt hat.
- MVars haben Locking eingebaut $\Rightarrow$ kein Busy-Wait!
- MVars sind *fair*: Wollen mehrere Threads gleichzeitig wiederholt schreiben/lesen, so werden kommen die Threads in der Reihenfolge Ihrer Anfragen abwechselnd zum Zug

Fallstricke:

- Implizite Synchronisation über gemeinsame Thunks
- MVars sind faul, d.h. können Thunks speichern
- Invariante „MVar voll“ kann durch Ausnahmen zerstört werden

Problemstellung

Manchmal ist es notwendig, laufende Threads zu unterbrechen:

- Benutzer klickt "Abbrechen" im UI-Thread
- Timeouts für diverse IO-Operationen

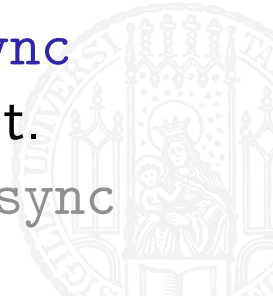
Eine Möglichkeit: Realisierung mit **Polling**, d.h. Thread fragt regelmässig MVar ab, ob Unterbrechung vorliegt.
Erfordert Disziplin und Sorgfalt, z.B. können Endlosschleifen auftreten, die auf Unterbrechung prüfen; erhöhte Last, etc.

Eine *andere Möglichkeit* sind **asynchrone Ausnahmen**:

```
throwTo :: Exception e => ThreadId -> e -> IO ()
```

Andere Threads an einer beliebigen Stelle zu unterbrechen ist auch nicht ganz unproblematisch. Modul `Control.Concurrent.Async` stellt uns aber eine relative einfache Schnittstelle als Hilfe bereit.

Paket: async



Async

```
data Async a = Async ThreadId (MVar Either SomeException a))

async :: IO a -> IO (Async a)
async action = do m <- newEmptyMVar
                  t <- forkIO (do r <- try action; putMVar m r)
                  return (Async t m)

cancel :: Async a -> IO ()
cancel (Async t _var) = throwTo t ThreadKilled

waitCatch :: Async a -> IO (Either SomeException a)
waitCatch (Async _ var) = readMVar var

wait :: Async a -> IO a
wait a = do r <- waitCatch a
            case r of Left e  -> throwIO e
                      Right a -> return a
```

Async

```
data Async a = Async ThreadId (MVar Either SomeException a))

async :: IO a -> IO (Async a)
async action = do m <- newEmptyMVar
                  t <- forkIO (do r <- try action; putMVar m r)
                  return (Async t m)

cancel :: Async a
cancel (Async t _

waitCatch :: Asy
waitCatch (Async

wait :: Async a -> IO a
wait a = do r <- waitCatch a
            case r of Left e  -> throwIO e
                      Right a -> return a
```

async ist ein bequemer Ersatz (bzw. Wrapper) für `forkIO`. Die `ThreadId` ist im Rückgabewert gespeichert; ebenso wird das Ergebnis oder die Ausnahme des neu eröffneten Threads nach dessen Beendigung in einer frischen `MVar` aufgesammelt.

Async

```
data Async a = Async ThreadId (MVar Either SomeException a))

async :: IO a -> IO (Async a)
async action = do m <- newEmptyMVar
                  t <- forkIO (do r <- try action; putMVar m r)
                  return (Async t m)

cancel :: Async a -> IO ()
cancel (Async t _var) = throwTo t ThreadKilled

waitCatch :: Asy
waitCatch (Async

wait :: Async a -> IO a
wait a = do r <- waitCatch a
            case r of Left e  -> throwIO e
                      Right a -> return a
```

`cancel` ist ein bequemer Weg, den richtigen Thread abzubrechen, da der Datentyp `Async` die `ThreadId` speichert

Async

```
data Async a = Async ThreadId (MVar Either SomeException a))

async :: IO a -> IO (Async a)
async action = do m <- newEmptyMVar
                  t <- forkIO (do r <- try action; putMVar m r)
                  return (Async t m)

cancel :: Async a -> IO ()
cancel (Async t _var) = throwTo t ThreadKilled

waitCatch :: Async a -> IO (Either SomeException a)
waitCatch (Async _ var) = readMVar var

wait :: Async a
wait a = do r <-
            case
                  Right a -> return a
```

waitCatch wartet, bis der abgezweigte Async-Thread beendet ist – entweder mit einer Ausnahme oder mit einem Ergebnis

Async

```
data Async a = Async ThreadId (MVar Either SomeException a))

async :: IO a -> IO (Async a)
async action = do m <- newEmptyMVar
                  t <- forkIO (do r <- try action; putMVar m r)
                  return (Async t m)

cancel :: Async a
cancel (Async t _

waitCatch :: Async a -> IO (Either SomeException a)
waitCatch (Async _ var) = readMVar var

wait :: Async a -> IO a
wait a = do r <- waitCatch a
            case r of Left e  -> throwIO e
                      Right a -> return a
```

wait wartet auf das Ende und Ergebnis eines Threads. Eine Ausnahme wird durch erneutes Werfen einfach nach aussen weitergereicht.

Async

```
data Async a = Async ThreadId (MVar Either SomeException a))

async :: IO a -> IO (Async a)
async action = do m <- newEmptyMVar
                  t <- forkIO (do r <- try action; putMVar m r)
                  return (Async t m)

cancel :: Async a -> IO ()
cancel (Async t _var) = throwTo t ThreadKilled

waitCatch :: Async a -> IO (Either SomeException a)
waitCatch (Async _ var) = readMVar var

wait :: Async a -> IO a
wait a = do r <- waitCatch a
            case r of Left e  -> throwIO e
                      Right a -> return a
```

Tatsächliche Implementierung weicht etwas ab und verwendet intern STM (siehe nächstes Kapitel).

Async

- `async :: IO a -> IO (Async a)` ersetzt letztendlich `forkIO` und sammelt das Ergebnis in einer neuen MVar
- `cancel :: Async a -> IO ()` ist ein bequemer Weg, den richtigen Thread abzubrechen, da der Datentyp `Async` ja auch die `ThreadId` speichert
- `waitCatch :: Async a -> IO (Either SomeException a)` wartet, bis der abgezweigte Async-Thread beendet ist – entweder mit einer Ausnahme oder mit einem Ergebnis
- `wait :: Async a -> IO a` wartet auf das Ergebnis eines Threads. Eine Ausnahme wird einfach nach aussen weitergereicht.

Beispiel

```
timeDownload :: String -> IO ()
timeDownload url = do
  (page, time) <- timeit $ getURL url
  printf "downloaded: %s (%d bytes, %.2fs)\n"
         url (B.length page) time

main = do
  as <- mapM (async . timeDownload) sites
  forkIO $ do                                -- Thread wartet auf q
     hSetBuffering stdin NoBuffering
     forever $ do c <- getChar
                  when (c == 'q') $ mapM_ cancel as
  rs <- mapM waitCatch as
  printf "%d/%d succeeded\n" (length (rights rs)) (length rs)
```

Verschiedene Threads werden gestartet, um Webseiten abzufragen.
Ein weiterer Thread wartet auf die Tastatureingabe `'q'`,
und unterbricht ggf. die anderen Threads.

Unterbrechung überall

Folgender Code soll eine Funktion auf eine MVar anwenden:

```
myModifyMVar_ :: MVar a -> (a -> IO a) -> IO ()
myModifyMVar_ m fkt = do
  a <- takeMVar m                                    -- <1>
  r <- fkt a `catch` \e -> do putMVar m a            -- <2>
                              throw e                -- <3>
  putMVar m r                                        -- <4>
```

Was passiert bei einer Unterbrechung durch einen anderen Thread?

Unterbrechung überall

Folgender Code soll eine Funktion auf eine MVar anwenden:

```
myModifyMVar_ :: MVar a -> (a -> IO a) -> IO ()
myModifyMVar_ m fkt = do
  a <- takeMVar m                                    -- <1>
  r <- fkt a `catch` \e -> do putMVar m a            -- <2>
                              throw e                -- <3>
  putMVar m r                                        -- <4>
```

Was passiert bei einer Unterbrechung durch einen anderen Thread?

Bei Unterbrechung zwischen <1> und <2> oder auch zwischen <2> und <4> bleibt die MVar leer!
Damit kann es zu Deadlocks in anderen Threads kommen, falls „mv nie leer“ als Invariante angenommen wurde!

Masking

```
mask :: ((forall a. IO a -> IO a) -> IO b) -> IO b
```

Funktion `mask` unterdrückt Ausnahmen temporär.

```
modifyMVar_ :: MVar a -> (a -> IO a) -> IO ()
modifyMVar_ m fkt = mask $ \restore -> do
  a <- takeMVar m
  r <- restore (fkt a) `catch` \e -> do putMVar m a
                                        throw e
  putMVar m r
```

- Eine maskierte Funktion kann nicht unterbrochen werden.
- Für die Ausführung von `(fkt a)` wird die Möglichkeit für Unterbrechungen temporär wiederhergestellt.
- Blockierte Aktionen (z.B. `takeMVar` und `putMVar`) können trotzdem unterbrochen werden!

Masking

Masking wird an verschiedenen Stellen eingesetzt, z.B. auch in der anfangs erwähnten `bracket` Funktion:

```
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
bracket before after thing = mask $ \restore -> do
    a <- before
    r <- restore (thing a) `onException` after a
    _ <- after a
    return r
```

Damit wird sichergestellt, dass die Aufräumaktionen in jedem Fall durchgeführt werden.

Masking

Das vorgestellte `async` war fehlerhaft gewesen, da im Falle einer Unterbrechung nicht sichergestellt war, dass die MVar gesetzt wird:

- nach dem `forkIO`, aber noch vor dem `try`
- nach dem `try`, aber vor `putMVar`

Auch hier hilft maskieren:

```
data Async a = Async ThreadId (MVar Either SomeException a))

async :: IO a -> IO (Async a)
async action = do
  m <- newEmptyMVar
  t <- forkIO (do r <- try action; putMVar m r)
  --Unterbrechung |hier und      |hier problematisch!
  return (Async t m)
```

Masking

Das vorgestellte `async` war fehlerhaft gewesen, da im Falle einer Unterbrechung nicht sichergestellt war, dass die MVar gesetzt wird:

- nach dem `forkIO`, aber noch vor dem `try`
- nach dem `try`, aber vor `putMVar`

Auch hier hilft maskieren:

```
data Async a = Async ThreadId (MVar Either SomeException a))

async :: IO a -> IO (Async a)
async action = do
  m <- newEmptyMVar
  t <- mask $ \restore -> forkIO (do r <- try (restore action);
                                     putMVar m r)
  return (Async t m)
```

Da `forkIO` immer die aktuelle Maskierung übernimmt, sind nun alle Lücken geschlossen!

forkFinally

```
async :: IO a -> IO (Async a)
async action = do
  m <- newEmptyMVar
  t <- mask $ \restore ->
         forkIO (do r <- try (restore action); putMVar m r)
  return (Async t m)
```

Da dieses Muster häufiger auftritt:

```
async :: IO a -> IO (Async a)
async action = do
  m <- newEmptyMVar
  t <- forkFinally action (putMVar m)
  return (Async t m)

forkFinally :: IO a -> (Either SomeException a -> IO ())
                                                -> IO ThreadId
forkFinally action fun =
  mask $ \restore ->
    forkIO (do r <- try (restore action); fun r)
```

Zusammenfassung Async

- Rein funktionaler Code kann jederzeit problemlos unterbrochen werden. Asynchrone Ausnahmen sind hier kein Problem.
  Im Gegensatz dazu:
  Unterbrechungen, welche jederzeit auftreten können, verursachen manchmal sehr schwer erkennbare Probleme für Code mit Seiteneffekten!
- Threads werden nie direkt terminiert. Jeder Thread darf immer noch Aufräumaktionen durchführen.
- Maskieren entspricht vorübergehenden Wechsel zu Polling. Ununterbrechbare Blöcke sollten nicht zu groß werden.
- Verwendung von durchdachten Abstraktionen wie Modul `Control.Concurrent.Async` können das Riskio reduzieren.

Motivation I

```
newtype Account = Account (MVar Int)

deposit  :: Account -> Int -> IO ()
deposit (Account a) n = modifyMVar_ a (\x -> return $ x+n)

withdraw :: Account -> Int -> IO ()
withdraw a n = deposit a (negate n)

transfer1 :: Int -> Account -> Account -> IO ()
transfer1 n from to = do  withdraw from n
                          -- Unterbrechung hier kritisch!
                          deposit  to   n
```

Problem: Trotz der eingebauten Locks könnte ein anderer Thread eine falsche Gesammtsumme sehen, wenn `transfer1` durch Block oder Ausnahme unterbrochen wurde.

Motivation II

```
newtype Account = Account (MVar Int)

deposit  :: Account -> Int -> IO ()
deposit (Account a) n = modifyMVar_ a (\x -> return $ x+n)

withdraw :: Account -> Int -> IO ()
withdraw a n = deposit a (negate n)

transfer2 :: Int -> Account -> Account -> IO ()
transfer2 n (Account from) (Account to) = do
  bal_from <- takeMVar from
  bal_to   <- takeMVar to
  putMVar from (bal_from - n)
  putMVar to   (bal_to   + n)
```

Problem: Vorheriges Problem kann immer noch auftreten; zusätzlich können auch noch Deadlocks entstehen!

Probleme mit Locking

In vielen anderen Sprachen muss man eigene explizite Locks verwenden. In Haskell auch machbar durch zusätzliche `MVar ()`, doch dadurch wird es nur schlimmer:

- Bei zu wenigen Locks drohen Race Conditions und verletzte Invarianten
- zu viele Locks schränken Nebenläufigkeit ein und verursachen Deadlocks
- Reihenfolge in der Locks gesetzt werden ist unklar, bzw. muss global festgelegt werden
- Platzierung der Locks oft unklar
- Ausnahmebehandlung kann schwierig sein, da alle Locks in einen konsistententen Zustand gebracht werden müssen

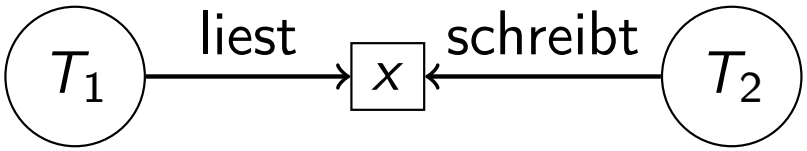
$\Rightarrow$ Fehleranfällige und globale Invarianten verhindern Modularität!

Transactional Memory

Idee Verschiedene Threads greifen auf gemeinsamem Speicher durch Transaktionen zu. Ein **Transaktion** ist ein Bündel von Speicherzugriffen eines Threads, welche **atomar** ausgeführt werden. Zwischenzustände werden für andere Threads unsichtbar.

Ausführung Transaktionen werden überwacht nebenläufig ausgeführt; bei einem Konflikt von Speicherzugriffen wird die Transaktion abgebrochen, alle bisherigen Effekte rückgängig gemacht und später wiederholt. $\Rightarrow$ keine Locks nötig!

Beispiel



T_1 *oder* T_2 abgebrochen, rückgängig gemacht & neu begonnen; andere Transaktion läuft weiter.

$\Rightarrow$ Ende 80er als Hardware für imperative Programme konzipiert; inzwischen teilweise verfügbar. z.B. Intel TSX seit 2012
Software Bibliotheken für sehr viele Sprachen verfügbar.

Software Transactional Memory (STM)

Rückspulen abgebrochener Transaktion in *funktionaler Welt* simpel.
Implementiert in Modul `Control.Concurrent.STM` Paket: `stm`

Monade `STM` bündelt Blöcke von Zugriffen auf gemeinsame Variablen (`TVar`) zu Transaktionen zusammen. Wie in jeder Monade können diese Blöcke miteinander kombiniert werden $\Rightarrow$ Modularität

`STM`-Transaktion können innerhalb der `IO`-Monade scheinbar atomar und *ohne Blockierung* (lock-free) ausgeführt werden. Probleme mit unpassenden Unterbrechungen sind von vornherein *ausgeschlossen*!

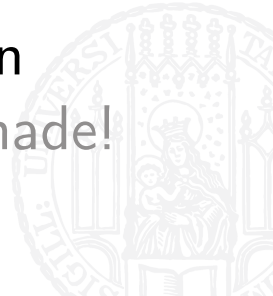
Innerhalb der `STM`-Monade ist kein `liftIO` möglich.
Dadurch sind alle Seiteneffekte bekannt und eine Berechnung kann bei Konflikten im Zugriff auf `TVar`s einfach zurückgespult werden.

TVar

```
newtype STM a = ... -- Eine Monade innerhalb von IO
data TVar a         -- Speicherzelle für Transaktionen

newTVar    :: a                    -> STM (TVar a)
readTVar   :: TVar a               -> STM a
writeTVar  :: TVar a -> a          -> STM ()
modifyTVar :: TVar a -> (a -> a) -> STM ()
swapTVar   :: TVar a -> a          -> STM a
```

- Eine `TVar` ist immer gefüllt und nie leer
- Weder `readTVar` noch `writeTVar` können jemals blockieren; `writeTVar` überschreibt immer
- Eine `TVar` wird in der Regel in verschiedenen `STM`-Aktionen verwendet – ganz im Gegensatz zu `IVars` der `Par`-Monade!
- Zugriffe auf `TVar` nur innerhalb der `STM` Monade möglich!



STM ausführen

Mehrere `TVar` Zugriffe werden monadisch zu einer Aktion der `STM`-Monade zusammengefasst (hier „Transaktion“ genannt), aber noch nicht ausgeführt.

Sofortiges *atomares* Ausführen einer Transaktion mit `IO` möglich:

```
atomically :: STM a -> IO a
```

Lese-/Schreibzugriff werden in einem Log festgehalten. Wenn die Transaktion beendet ist, wird das Log geprüft:

Konflikt „roll-back“: Log verwerfen, Transaktion neu starten

Okay Alle Schreibzugriffe ausführen, zu beschreibende `TVars` werden vorübergehenden gesperrt

⇒ `STM`-Transaktion sollten kurz und nicht unendlich sein.

Beispiel STM

```
-- in Thread 1:
atomically $ do
    v <- readTVar acc
    writeTVar acc (v + 1)

-- in Thread 2:
atomically $ do
    v <- readTVar acc
    writeTVar acc (v - 3)
```

Hier kann nichts schief gehen, da sichergestellt wird, das Lese- und Schreibzugriff immer komplett oder gar nicht durchgeführt werden.

Beispiel: Wird Thread 1 vor dem Schreiben unterbrochen und Thread 2 verändert die `TVar` zwischenzeitlich, dann fängt Thread 1 mit der atomaren `STM`-Transaktion einfach wieder von vorne an.

Beispiel STM

```
transfer3 :: Int -> TVar Int -> TVar Int -> STM ()
transfer3 n from to = do
  bal_from <- readTVar from
  bal_to   <- readTVar to
  writeTVar from (bal - n)
  writeTVar to   (bal + n)
```

Falls während einer Unterbrechungen die betroffenen `TVar` verändert wurden, dann wird die gesamte Aktion später wiederholt. Ein einzelner `writeTVar` kann nie stattfinden!

Explizites Blocking mit retry

Funktion `retry :: STM a` löst explizites roll-back aus.

Der Thread wird angehalten, bis sich mindestens eine der bis dahin *gelesenen* `TVar` verändert hat. Deadlocks dadurch möglich

Dies ist hilfreich wenn im Programm erkannt wird, dass eine Transaktion nicht erfolgreich abgeschlossen werden kann.

Beispiel: Abheben von einem leeren Konto

```
withdrawP :: TVar Int -> Int -> STM ()
withdrawP acc n = do
  bal <- readTVar acc
  if bal < n
    then retry
    else writeTVar acc (bal - n)
```

Komposition mit orElse

`orElse :: STM a -> STM a` erlaubt alternative Auswahl zwischen zwei `STM`-Transaktionen: Schlägt die erste Transaktion fehl mit `retry`, so wird die zweite Transaktion ausgeführt. Wenn beide Transaktionen fehlschlagen, schlägt auch die gesamte Transaktion fehl und wird komplett wiederholt.

Beispiel: Transfer von zwei Konto Alternativen

```
transEither :: Int -> TVar Int
                   -> TVar Int -> TVar Int ->  STM ()
transEither n from1 from2 to3 = do
    (withdrawP from1 n  `orElse`  withdrawP from2 n)
    deposit acc3 n
```

Beobachtung: `retry` startet immer ganz am Anfang einer `STM`-Aktion, auch bei Komposition mehrerer Aktionen!

STM Monade

- `atomically :: STM a -> IO a`
  `STM`-Aktion mit `TVar`-Zugriffen sofort atomar ausführen, d.h. kein anderer Thread sieht einen Zwischenzustand bei sämtlichen *berührten* `TVar`-Variablen.

- `retry :: STM a`
  Aktuelle `STM`-Aktion abbrechen und später von vorn wiederholen. Der Thread wird blockiert, bis sich mindestens eine gelesene `TVar` verändert hat $\Rightarrow$ explizites Blocking

- `orElse :: STM a -> STM a -> STM a`
  Komposition. Wert des ersten Arguments, falls kein `retry` auftrat, sonst zweites Argument.
  Tritt auch dort ein `retry` auf, wird wieder das erste versucht.

Beispiel: Gepufferte Kanäle

```
newtype TList a = TList (TVar [a])

newTList :: STM (TList a)
newTList = do v  <- newTVar []
              return (TList v)

writeTList :: TList a -> a -> STM ()
writeTList (TList v) a = do list <- readTVar v
                            writeTVar v (list ++ [a])

readTList :: TList a -> STM a
readTList (TList v) = do  xs <- readTVar v
                          case xs of
                            []     -> retry  -- blocks if empty
                            (x:xs') -> do
                              writeTVar v xs'
                              return x
```

Verbesserung durch Einsatz zweier Listen als Warteschlange möglich, dann recht effizient siehe „Purely Functional Datastructures“ von C.Okasaki; auch im Buch von S.Marlow

Beispiel: TMVar

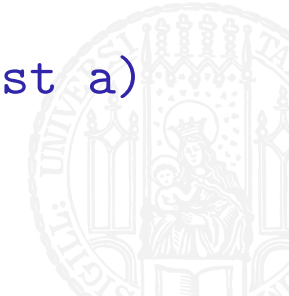
Blockierende Variablen in STM:

```
newtype TMVar a = TMVar (TVar (Maybe a))

newEmptyTMVar :: STM (TMVar a)
newEmptyTMVar = TMVar <$> newTVar Nothing

takeTMVar :: TMVar a -> STM a
takeTMVar (TMVar t) = do m <- readTVar t
                         case m of
                           Nothing -> retry -- block
                           Just a  -> do writeTVar t Nothing
                                         return a
putTMVar :: TMVar a -> a -> STM ()
putTMVar (TMVar t) a = do m <- readTVar t
                          case m of
                            Nothing -> writeTVar t (Just a)
                            Just _  -> retry -- block
```

. . . wird bereits bereitgestellt von Control.Concurrent.STM



Beispiel: TMVar

Blockierende Variablen in `STM`:

```
newtype TMVar a = TMVar (TVar (Maybe a))
```

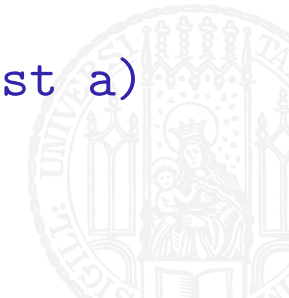
Achtung, Deadlocks hier wieder leicht möglich wie gehabt:

```
tmv <- newEmptyTMVar
x <- takeTMVar tmv
```

```
takeTMVar :: TMVar a -> STM a
takeTMVar (TMVar t) = do m <- readTVar t
                         case m of
                           Nothing -> retry -- block
                           Just a  -> do writeTVar t Nothing
                                         return a

putTMVar :: TMVar a -> a -> STM ()
putTMVar (TMVar t) a = do m <- readTVar t
                          case m of
                            Nothing -> writeTVar t (Just a)
                            Just _  -> retry -- block
```

. . . wird bereits bereitgestellt von `Control.Concurrent.STM`



Ausnahmebehandlung in STM

Ausnahmen brechen eine `STM`-Aktion einfach ab.
Dank roll-back ist dies immer unproblematisch.

Dennoch ist es möglich eine Ausnahmebehandlung durchzuführen:

```
throwSTM :: Exception e => e -> STM a
catchSTM :: Exception e => STM a -> (e -> STM a) -> STM a
```

Diese funktionieren wie in der `IO`-Monade, mit dem Unterschied, dass auch bei `catchSTM` alle Seiteneffekte des ersten Arguments verworfen werden, bevor der Handler ausgeführt wird.

Probleme bei STM

Mögliche Probleme bei Verwendung der `STM`-Monade:

- `STM`-Aktionen sollten nicht zu groß werden,
  da ggf. alles wiederholt werden muss
- Im Gegensatz zu `MVar`-Code kein faires Scheduling: es werden immer alle auf eine `TVar` wartenden Threads geweckt da unbekannt ist, welche Konditionen genau zum blocking geführt haben.
  Kürzere `STM`-Aktionen können deshalb schnell erfolgreich abschließen und dadurch längere `STM`-Aktionen auf gleicher `TVar` zum ewigen roll-back zwingen
- Langsamer als `MVar`-Code: `readTVar` hat Laufzeit linear in der Anzahl der Zugriffe, da bei jedem Zugriff das `TVar`-Transaktionen Log der Aktion geprüft werden muss

Fallstrick

Was ist besser?

1. `atomically $ mapM takeTMVar tmvs`
2. `mapM (atomically . takeTMVar) tmvs`

Fallstrick

Was ist besser?

1. `atomically $ mapM takeTMVar tmvs`
2. `mapM (atomically . takeTMVar) tmvs`

Unterschiedliche Semantik

1. Alle Werte waren gleichzeitig gültig — `mapM` in `STM` ausgeführt
2. Werte von verschiedenen Zeitpunkten — `mapM` in `IO` ausgeführt

Fallstrick

Was ist besser?

1. `atomically $ mapM takeTMVar tmvs`
2. `mapM (atomically . takeTMVar) tmvs`

Unterschiedliche Semantik

1. Alle Werte waren gleichzeitig gültig — `mapM` in `STM` ausgeführt
2. Werte von verschiedenen Zeitpunkten — `mapM` in `IO` ausgeführt

Unterschiedliche Performanz

1. $O(n^2)$ Sobald eine `TMVar` gefüllt wird, müssen alle vorangegangenen auch neu eingelesen werden!
2. $O(n)$ Jede Variable wird einzeln für sich eingelesen.

Zusammenfassung STM

- `STM` erlaubt "lock-free" Code auf vielen gemeinsamen Variablen
  Dies erlaubt *hohe Modularität*, da keine globalen Invarianten zur Vermeidung von Deadlocks erzwungen werden müssen
- `STM`-Monade bietet gute Nebenläufigkeit, da ein roll-back nur auftritt, wenn auch tatsächlich gemeinsam benutzte Variablen verändert werden.
  Zwei `STM`-Aktionen auf unterschiedlichen `TVar` können sich nie gegenseitig stören!
- `retry` blockiert einen Thread, bis relevante `TVar` verändert wurde $\Rightarrow$ gute Modularität!
- **Software Transactional Memory** gibt es inzwischen für viele (imperative) Sprachen,

STM Operationen

```
atomically :: STM a -> IO a  -- STM Aktion atomar ausfuehren
retry      :: STM a          -- STM Aktion wiederholen
check      :: Bool -> STM () -- Wiederholen falls falsch
orElse     :: STM a -> STM a -> STM a
 -- Komposition: Wert des ersten Arguments,
 -- falls kein retry auftrat, sonst zweites Argument oder retry

throwSTM :: Exception e => e -> STM a
catchSTM :: Exception e => STM a -> (e -> STM a) -> STM a

newTVar     :: a -> STM (TVar a)
newTVarIO   :: a -> IO  (TVar a) -- == atomically . newTVar
readTVar    :: TVar a -> STM a
readTVarIO  :: TVar a -> IO a -- besser als atomically.readTVar
writeTVar   :: TVar a -> a -> STM()
modifyTVar  :: TVar a -> (a -> a) -> STM ()
modifyTVar' :: TVar a -> (a -> a) -> STM () -- strict
```

Zusammenfassung Explizite Parallelität

- Echte explizite Nebenläufigkeit möglich wie in anderen Sprachen auch, mit den gleichen Problemen, jedoch Abhilfe durch modulare Bibliotheken
- Implizite Synchronisation über gemeinsame Teilausdrücke
- Explizite Synchronisation über `MVar` oder `Chan` mit eingebautem locking
- "lock-free" Code auf gemeinsam `TVars` in `STM`-Monade

Quellenangabe

Dieses Kapitel zeigte Ideen und Code-Beispiele primär aus folgenden Quellen:

- Simon Marlow. "Parallel and Concurrent Programming in Haskell". O'Reilly, July 2013, ISBN 978-1-449-33594-6
- Simon Peyton-Jones. "Beautiful Concurrency"
  Kapitel aus "Beautiful Code", edited by Greg Wilson and Andy Oram. O'Reilly, July 2007, ISBN 978-0-596-51004-6
- Frühere Skripe des Lehrstuhls TCS der LMU München unter anderem von Hans-Wolfgang Loidl und Andreas Abel.