

FORTGESCHRITTENE FUNKTIONALE PROGRAMMIERUNG MIT HASKELL

AUSNAHMEN & NEBENLÄUFIGKEIT

Steffen Jost

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

28. November 2014

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 kann.



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 | Kopfweg | 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 Kopfweg = Right nimmAspirin -- Fehlerbehandlung
```

```
hilfe anderes = Left  anderes -- andere Leiden nicht abgefangen
```

NACHTEILE: Die Monade kann manchmal 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 implizite auftretende Ausnahmen, welche wir leider nicht am Typ erkennen können:

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

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 :: 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

```

{-# LANGUAGE DeriveDataTypeable #-}
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 Standardbibliothek 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
```



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 leicht um benutzerdefinierbare Ausnahmen erweitert werden können
- Ausnahmen nur gezielt abfangen um
 - 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

⇒ 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**

Im Gegensatz zu GpH sind diese IO-Threads explizite Objekte, die im Code kontrolliert werden.

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

`forkIO` erzeugt einen IO-Thread, der mittels einer `ThreadId` identifiziert wird.

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.



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 auszuführen: \Rightarrow Fair Scheduling

BBBBABABABABABABABABABABABABABABABABABABAABABABABA...



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.
⇒ 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
          putStrLn "Done."
```

Das Programm wird beendet, sobald der Hauptthread beendet wird.
⇒ Notwendigkeit zur Synchronisation zwischen Threads



IMPLIZITE SYNCHRONISATION

```
import Control.Concurrent

n = fib 42
fib n | n < 2      = 1
      | otherwise = fib (n-1) + fib (n-2)

main = do putStrLn "Creating Threads."
          forkIO $ print ("A",n)
          forkIO $ print ("B",n)
          print ("C",n)
          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 nur innerhalb der IO-Monade möglich
- 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

```
import Control.Concurrent
import Control.Concurrent.MVar

task :: MVar Int -> MVar Int -> IO ()
task receive send = do -- some work
  v <- takeMVar receive
  let res = 3 * v
  putMVar send res

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

Die Threads warten aufeinander, bis die Werte bereitstehen.



BEISPIEL FÜR DEADLOCK

```
task :: MVar Int -> MVar Int -> IO ()
task receive send = do -- some work
  v <- takeMVar receive
  let res = 3 * v
  putMVar send res

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
```

ASYNCHRONES 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.
```



EXPLIZITE SYNCHRONISATION: MVars

MVars erlauben direkt

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

Da MVars für beliebige Typen deklariert werden können, ist es relativ leicht möglich, Kommunikationskanäle mit unbegrenzten Buffer (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

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`



FALLSTRICKE 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 `(>>=)` garantiert lediglich, dass Seiteneffekte Reihenfolge einhalten:

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



ZUSAMMENFASSUNG MVAR

- MVars haben einen Locking-Mechanismus eingebaut.
⇒ kein Busy-Wait!
- 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 sind *fair*: Wollen mehrere Threads gleichzeitig wiederholt schreiben/lesen, so werden kommen die Threads in der Reihenfolge Ihrer Anfragen abwechselnd zum Zug
- MVars sind faul



FORTGESCHRITTENE FUNKTIONALE PROGRAMMIERUNG MIT HASKELL

AUSNAHMEN & NEBENLÄUFIGKEIT

Steffen Jost

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

5.12.2014



PROBLEMSTELLUNG

Manchmal ist es notwendig, laufende Threads zu unterbrechen:

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

Eine Möglichkeit, dies zu realisieren, besteht in **Polling**: 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 :: e => ThreadId -> e -> IO ()
```

Auch diese sind nicht ganz unproblematisch.

Modul `Control.Concurrent.Async` stellt eine relative einfach Schnittstelle als Hilfe bereit.



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

- `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 abzurechnen, da der Datentyp `Async` ja auch die `ThreadId` speichert
- `waitCatch :: Async a -> IO (Either SomeException a)` wartet, bis der abgezwigte 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
    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:

```
modifyMVar_ :: MVar a -> (a -> IO a) -> IO ()
modifyMVar_ mv f = do
  a <- takeMVar m                                -- <1>
  r <- f a `catch` \e -> do                      -- <2>
    putMVar m a
    throw e
  putMVar m r                                    -- <3>
```

Was passiert bei einer Unterbrechung durch einen anderen Thread?



UNTERBRECHUNG ÜBERALL

Folgender Code soll eine Funktion auf eine MVar anwenden:

```

modifyMVar_ :: MVar a -> (a -> IO a) -> IO ()
modifyMVar_ mv f = do
  a <- takeMVar m                                -- <1>
  r <- f a `catch` \e -> do                      -- <2>
                                                putMVar m a
                                                throw e
  putMVar m r                                    -- <3>

```

Was passiert bei einer Unterbrechung durch einen anderen Thread?

Bei Unterbrechung zwischen <1> und <2> oder auch zwischen <2> und <3> bleibt die MVar leer.

Damit kann es zu Deadlocks in anderen Threads kommen!



MASKING

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

Die Funktion `mask` unterdrückt Ausnahmen temporär.

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

- Eine maskierte Funktion kann nicht unterbrochen werden.
- Für die Ausführung von `f a` wird die Möglichkeit für Unterbrechungen temporär wiederhergestellt
- Blockierte Aktionen (z.B. `takeMVar` und `putMVar`) dürfen 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 eine Unterbrechung nicht sichergestellt war, dass die `MVar` gesetzt wird:

- nach dem `fork`, 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` ebenfalls 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 vorübergehendem Wechsel zu Polling. Ununterbrechbare Blöcke dürfen aber nicht zu groß werden.
- Verwendung von durchdachten Abstraktionen wie Modul `Control.Concurrent.Async` können das Risiko reduzieren.



MOTIVATION

```

newtype Account = Account (MVar Int)

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

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

transfer1 :: Int -> Account -> Account -> IO ()
transfer1 n from to = do  withdraw from n
                          deposit  to  n

```

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



MOTIVATION

```

newtype Account = Account (MVar Int)

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

deposit  :: Account -> Int -> IO ()
deposit a n = withdraw 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** können Nebenläufigkeit einschränken und Deadlocks verursachen
- **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 konsistenten Zustand gebracht werden müssen

⇒ Fehleranfällige und globale Invarianten verhindern Modularität!



SOFTWARE TRANSACTIONAL MEMORY

Software Transactional Memory (STM) aus Paket `Control.Concurrent.STM` erlaubt die Ausführung von IO-Threads auf gemeinsamen Variablen (**TVar**) **ohne** Blockierung (“lock-free”).

Stattdessen wird die rein funktionale Natur ausgenutzt: Falls während einer Berechnung Konflikte im Zugriff auf **TVars** auftreten, wird die gesamte Berechnung einfach “zurückgespult”.

Dazu müssen alle Operationen auf **TVar** innerhalb der **STM** Monade ausgeführt werden. Damit sind alle Seiteneffekte bekannt, um sicher “zurückspulen” zu können.

Es werden dann Blöcke kreiert, welche atomar ausgeführt werden. Probleme mit unpassender Unterbrechungen sind dabei von vornherein ausgeschlossen!



TVar

```
newtype STM a = ... -- Eine Monade
```

```
data TVar a -- Speicherzelle für atomare Zugriffe
```

```
newTVar    :: a -> STM (TVar a)
```

```
readTVar   :: TVar a -> STM a
```

```
writeTVar  :: TVar a -> a -> STM()
```

- Eine `TVar` ist immer gefüllt und nie leer
- Weder `readTVar` noch `writeTVar` können jemals blockieren



TVar

```
newtype STM a = ... -- Eine Monade
```

```
data TVar a -- Speicherzelle für atomare Zugriffe
```

```
newTVar    :: a -> STM (TVar a)
```

```
readTVar   :: TVar a -> STM a
```

```
writeTVar  :: TVar a -> a -> STM()
```

- Eine `TVar` ist immer gefüllt und nie leer
- Weder `readTVar` noch `writeTVar` können jemals blockieren

Frage: Wie kann das funktionieren bei nebenläufigen Zugriffen auf die gleiche Variable?

Antwort: Zugriffe finden nur innerhalb der `STM` Monade statt!



STM MONADE

- `atomically :: STM a -> IO a`
STM-Aktion mit mehreren `TVar`-Zugriffen atomar ausführen, d.h. kein anderer Thread sieht einen Zwischenzustand bei den berührten `TVar`-Variablen.
- `retry :: STM a`
Aktuelle `STM`-Aktion abbrechen und später wiederholen. Der Thread wird solange 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 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 atomischen STM-Aktion 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` ermöglicht ein explizites roll-back auszulösen.

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

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

Funktion `orElse` ermöglicht es eine Auswahl zwischen zwei `STM`-Aktionen.

Wenn eine Transaktion fehlschlägt, so wird die andere ausgeführt. Nur wenn beide Transaktionen fehlschlagen, schlägt auch die gesamte Transaktion fehl und wird 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!



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
                          (x:xs') -> do
                            writeTVar v xs'
                            return x
```

Verbesserung durch Einsatz zweier Listen als Warteschlange
möglich, dann sehr effizient

siehe Buch von S.Marlow



AUSNAHMEBEHANDLUNG IN STM

Ausnahmen brechen eine **STM**-Aktion einfach ab.
Dank dem Rollback 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
- kein faires Scheduling, da alle auf eine TVar wartenden Threads geweckt werden müssen – es ist ja unbekannt, welche Konditionen genau zum blocking geführt haben. Kürzere STM-Aktionen können so immer schneller erfolgreich abschliessen und davon abhängige längere STM-Aktionen zum Rollback zwingen
- readTVar / writeTVar haben Laufzeit linear in der Anzahl der Zugriffe, da bei jedem Zugriff das TVar-Transaktionen Log der Aktion geprüft werden muss



ZUSAMMENFASSUNG STM

- **STM**-Monade 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 dann auftritt, wenn auch tatsächlich gemeinsam benutzte Variablen verändert werden.
Zwei **STM**-Aktionen auf unterschiedlichen **TVar** können sich nicht gegenseitig stören!
- **retry** blockiert einen Thread automatisch, bis eine relevante **TVar** verändert wurde.
Auch hier ist gute Modularität gewährleistet!



STM OPERATIONEN

```

atomically :: STM a -> IO a -- STM Aktion atomar ausführen
retry      :: STM a          -- STM Aktion wiederholen
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)
readTVar  :: TVar a -> STM a
readTVarIO :: TVar a -> IO a --schneller 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. O’Reilly, July 2007, ISBN 978-0-596-51004-6
- Frühere Skripte des Lehrstuhls TCS der LMU München unter anderem von Hans-Wolfgang Loidl und Andreas Abel.

