LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Fortgeschrittene Funktionale Programmierung

Wintersemester 2021/22

10 Parallelität und Nebenläufigkeit in Haskell

Prof. Dr. David Sabel

LFE Theoretische Informatik

Letzte Änderung der Folien: 27. Januar 2022 ♦ Die Folien basieren zum Teil auf Material von Dr. Steffen Jost, dem an dieser Stelle für die Verwendungserlaubnis herzlich gedankt sei.

Parallelität und Nebenläufigkeit

Parallelität

- Ziel: Schnelle Ausführung von Programmen durch gleichzeitige Verwendung mehrerer Prozessoren
- Funktionale Programmiersprachen bieten „high-level"-Ansätze, während in anderen Programmiersprachen oft mehr „low-level"-Primitive zu finden sind.
- Beachte: Bisher kein funktionierender Ansatz zur automatischen Parallelisierung
- Folge: Programmierer muss manuell parallelisieren und prüfen (testen), ob das Programm beschleunigt

Nebenläufigkeit

- Mehrere Berechnungen werden nichtdeterministsch / verzahnt ausgeführt
- Evtl. auch nur auf einem Prozessor, evtl. auch parallel
- Ziel neben Beschleunigung ist die quasi gleichzeitige Ausführung mehrerer Aufgaben, wie Tastatureingaben, Mausklick, Anfragen an Webserver

Überblick

In diesem Kapitel:

Ansätze zur parallelen Programmierung und zur nebenläufigen Programmierung in Haskell:

- Glasgow parallel Haskell
- Programmieren mit der Par-Monade
- Concurrent Haskell
- STM-Haskell

Grundlegendes

Häufiges **Vorgehen zum Parallelisieren**:

- Nehme sequentiellen Algorithmus
- Parallelisiere den sequentiellen Algorithmus
- Unterteile Berechnung in sinnvoll unabhängige Einheiten
- Berechne die Einheiten parallel
- Setze das Gesamtergebnis zusammen

Amdahls Gesetz:

$$\text{maximale Beschleunigung} \leq \frac{1}{(1-P) + P/N}$$

wobei N die Anzahl Kerne und P der parallelisierbare Anteil des Programms ist

Z.B. 80% parallelisierbar $\rightarrow$ maximal mögliche Beschleunigung: 5

Sprechweisen

- **Thread** = Ausführungsstrang eines Programms, der sequentiell abläuft.
- **Programm / Prozess** kann aus mehreren Threads bestehen
- **HEC** = Haskell Execution Context ist ein virtueller Core/Thread in Haskell
- **Core**= Prozessorkern, kann einen Thread gleichzeitig abarbeiten,
  üblicherweise: mehr Threads als Cores!
- **Scheduling** = Betriebssystem / Run-Time-System verteilt Threads auf die Prozessorkerne (Threads abwechselnd, stückchenweise laufen lassen)
- **Mapping** = Zuordnung Threads zu Prozessoren

Programmieren mit Threads

- Ein Thread erledigt Teilaufgabe
- Threads **synchronisieren**, wenn sie voneinander abhängen
- **Race-Condition**: Gesamtergebnis der Berechnung hängt von der Ausführung der Threads ab und ist nicht mehr vorhersagbar.
- **Deadlock** (Verklemmung): Ein Thread wartet auf das Ergebnis eines anderen Threads, welcher direkt oder indirekt selbst auf das Ergebnis des ersten Threads wartet.

In nebenläufigen Programmen:
Probleme **nicht vermeidbar**

In parallelen Programmen in rein funktionalen Programmen:
Probleme können **eliminiert werden**
(wegen referentieller Transparenz ist die Auswertungsreihenfolge nahezu beliebig)

Parallele rein funktionale Programme

Beachte: Parallelisierung darf das **Terminierungsverhalten nicht ändern**!

Z.B. `const 1 bot`, wobei

```
const x y = x
bot = bot
```

Wird `bot` parallel ausgewertet und die Gesamtauswertung endet erst,
wenn alle parallelen Auswertungen beendet sind, dann ändert sich das Terminierungsverhalten gegenüber dem sequentiellen Programm!

Daher besser: **Kann-Parallelismus** statt **Muss-Parallelismus**

GHC-Multithreading

Einstellen der verwendbaren Kerne (*number*) im GHC:

- Statisch während des Kompilierens
  `ghc -threaded with-rtsopts="-N`*number*`"`
- Als Kommandozeilenparameter zur Ausführung
  `ghc MyProg.hs -threaded -rtsopts # kompileren`
  `./MyProg +RTS -N`*number*
- Dynamisch im Programm

```
import Control.Concurrent
-- setNumCapabilities :: Int -> IO ()
...
  setNumCapabilities number
...
```

  Kompilieren mit `-threaded`

Profiling

GHC erlaubt **Profiling**: Protokolliere, was das Programm wie lange macht.

- Spezielles Kompilieren mit `-prof -fprof-auto -rtsopts`

  ```
  > ghc MyProg.hs -O2 -prof -fprof-auto -rtsopts
  ```

- Anschließend wird das Programm mit der RTS-Option `-p` ausgeführt:

  ```
  > ./MyProg +RTS -p
  ```

- Erstellt `MyProg.prof`, dort kann man sehen, wie viel Zeit wofür aufgewendet wurde
- Bibliotheken mit Profiling installieren

  ```
  > cabal install mein-modul -p  # bei cabal
  > stack build --profile        # bei Verwendung von stack
  > stack run --profile -- Main +RTS -p
  ```

- Viele Optionen verfügbar (siehe Dokumentation)

Einfacher: Statistikausgabe des Runtime-Systems

- Option `-s` für das Laufzeitsystem, d.h.

  ```
  > ghc -O -rtsopts MyProg.hs
  > ./MyProg +RTS -s
  ```

- Druckt Statistik zu
  - Laufzeit
  - Platzbedarf
  - Garbage Collection
  - Eckdaten zur parallelen Auswertung
- Mit `-s`*Dateiname* Ausgabe in eine Datei schreiben

Threadscope für Parallele Programme

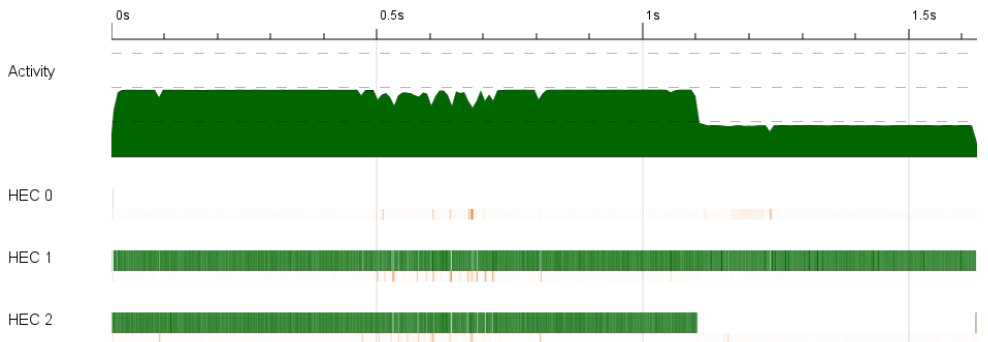
- Threadscope-Profiler (https://wiki.haskell.org/ThreadScope)
- Threadscope installieren (siehe Wiki-Seite)
  - z.B. mit `stack install threadscope`
  - oder binary herunterladen

  Verwendung

  ```
  > ghc MyPrg.hs -threaded -eventlog -rtsopts
  > ./MyPrg +RTS -N4 -l
  > threadscope MyPrg.eventlog
  ```

- D.h. kompileren mit `-threaded`, `-eventlog` und `-rtsopts`
- Danach visualisiert Threadscope visualisiert dann das Event-Log.

Threadscope-Ansicht



Anfangs werden zwei Kerne genutzt (HEC1, HEC2), dritter Kern (HEC0) bieb ungenutzt.
Arbeitsphasen sind in Grün dargestellt, Garbage-Collection des Laufzeitsystems in Orange.

Semi-explizite Parallelität:

Glasgow parallel Haskell

Glasgow parallel Haskell

- bereits 1996 entwickelt
- Modul `Control.Parallel`, erweitert Haskell durch zwei Primitive:

```
par :: a -> b -> b
pseq :: a -> b -> b
```

- `x `par` e` erlaubt die parallele Auswertung von `x` und `e`.
- Kann-Parallelismus:
  Laufzeitsystem bekommt „Hinweis“, dass `x` parallel zu `e` ausgewertet werden kann
- Es wird ein Spark für `x` erzeugt
- Laufzeitsystem verwaltet Sparks: Werden zur WHNF ausgewertet, falls Prozessor frei
- `x `pseq` e`: Sequentielle Auswertung von `x` und `e` bis zur WHNF
- Unterschied zu `seq`: Kompiler garantiert bei `pseq` die sequentielle Auswertung

Beispiel: Exponentielle Berechnung der Fibonacci-Zahlen

```
import Control.Parallel
import System.Environment(getArgs)

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

main = do
  (inp1:inp2:_) <- getArgs
  let x = fib (read inp1)
  let y = fib (read inp2)
  let s = x `par` y `pseq` x+y  -- parallele Berechnung zweier Werte
  print s
```


Beispiel: Exponentielle Berechnung der Fibonacci-Zahlen (2)

Compilieren und Ausführen:

```
> stack ghc -- fib1.hs -O  -threaded -rtsopts
./fib1 41 41 +RTS -N1 -s  2>&1 | sed -n '/Total/p'
  Total   time   17.953s  ( 17.991s elapsed)
> ./fib1 41 41 +RTS -N2 -s  2>&1 | sed -n '/Total/p'
  Total   time   20.591s  ( 10.341s elapsed)
```

- Mit 2 Kernen wird nur noch ungefähr die Hälfte der Zeit benötigt
- Summe der Rechenzeit ist gestiegen.
- Verwaltung der Sparks kostet Zeit

Beispiel: Exponentielle Berechnung der Fibonacci-Zahlen (3)

Mehr parallelisieren:

```
import Control.Parallel
import System.Environment(getArgs)

pfib :: Integer -> Integer
pfib n | n < 2     = 1
       | otherwise = let fn1 = pfib (n-1)
                         fn2 = pfib (n-2)
          in fn1 `par` fn2 `pseq` fn1 + fn2    -- parallel fn1 berechnen!

main = do
  (inp1:inp2:_) <- getArgs
  let x = pfib (read inp1)
  let y = pfib (read inp2)
  let s = x `par` y `pseq` x+y
  print s
```


Beispiel: Exponentielle Berechnung der Fibonacci-Zahlen (3)

Ausführung:

```
> ./fib2 41 41 +RTS -N1 -s  2>&1 | sed -n '/Total\|SPARKS/p'
   SPARKS: 535828591 (0 converted, 234141903 overflowed, 0 dud, 301642185 GC'd, 44503 fizzled)
  Total   time   21.857s  ( 21.921s elapsed)
> ./fib2 41 41 +RTS -N2 -s  2>&1 | sed -n '/Total\|SPARKS/p'
  SPARKS: 535893347 (48 converted, 226501882 overflowed, 0 dud, 309088071 GC'd, 303346 fizzled)
  Total   time   26.268s  ( 13.191s elapsed)
> ./fib2 41 41 +RTS -N3 -s  2>&1 | sed -n '/Total\|SPARKS/p'
  SPARKS: 535954678 (97 converted, 223541630 overflowed, 0 dud, 311958286 GC'd, 454665 fizzled)
  Total   time   30.573s  ( 10.221s elapsed)
> ./fib2 41 41 +RTS -N4 -s  2>&1 | sed -n '/Total\|SPARKS/p'
  SPARKS: 536058575 (178 converted, 217787349 overflowed, 0 dud, 317527048 GC'd, 744000 fizzled)
  Total   time   33.098s  (  8.301s elapsed)
> ./fib2 41 41 +RTS -N5 -s  2>&1 | sed -n '/Total\|SPARKS/p'
  SPARKS: 536152118 (293 converted, 159965617 overflowed, 0 dud, 375147985 GC'd, 1038223 fizzled)
  Total   time   41.048s  (  8.351s elapsed)
> ./fib2 41 41 +RTS -N6 -s  2>&1 | sed -n '/Total\|SPARKS/p'
  SPARKS: 536205153 (352 converted, 109976066 overflowed, 0 dud, 424988603 GC'd, 1240132 fizzled)
  Total   time   49.862s  (  8.362s elapsed)
> ./fib2 41 41 +RTS -N7 -s  2>&1 | sed -n '/Total\|SPARKS/p'
  SPARKS: 538082452 (1675 converted, 116996337 overflowed, 0 dud, 417045295 GC'd, 4039145 fizzled)
  Total   time   61.974s  (  9.061s elapsed)
```

- Laufzeit verbessert sich nur unwesentlich oder verschlechtert sich
- Zuviel Overhead durch zu viele Sparks!

Sparks

Ein Spark steht für eine mögliche parallele Berechnung.

Zur Laufzeit gibt es mehrere Möglichkeiten für einen Spark:

- Konvertiert (converted): Der Spark wurde ausgerechnet.
- Abgeschnitten (pruned): Der Spark wurde nicht ausgerechnet.
  - Dud: Der Spark war schon ein ausgerechneter Wert.
  - Fizzled: Inzwischen durch anderen Thread berechnet.
  - Garbage Collected: Keine Referenz zum Spark, Wert wird nicht mehr benötigt.
  - Overflowed: Der Spark wurde gleich verworfen, da der Ringpuffer der Sparks voll ist.

Idealerweise: Mehr Sparks konvertiert als abgeschnitten.

Sparks (2)

- Spark ist sehr billig (deutlich einfacher als ein Thread)
- Spark-Pool ist ein Ringpuffer von Thunks
- Akzeptabel: Mehr Sparks einzuführen, als letztendlich ausgeführt werden
- Dennoch schädlich, wenn zu viele Sparks angelegt werden
  (z.B. wenn die Ausführung des Sparks schneller geht als das Anlegen des Sparks selbst).
- Granularität (Größe der einzelnen Aufgaben/Threads/Sparks)
  - nicht zu klein (Verwaltungsaufwand)
  - nicht zu groß sein (Auslastung)

Verzögerte Auswertung

Problem für parallele Auswertung durch verzögerte Auswertung:

```
genlist :: Integer -> Integer -> [[Integer]]
genlist _ 0 = []
genlist x n = let f = fib x
                  h = (:) f []
                  t = genlist x (n-1)
              in h `par` t `pseq` h : t
main = do
  let l = genlist 38 6
  mapM_ print l
```

- Liste wird parallel zusammengesetzt
- Aber Fibonacci-Berechnungen finden erst bei `print` statt.

Echten Speedup mit:

```
...
              in f `par` t `pseq` h : t  -- parallel f berechnen
...
```


Verzögerte Auswertung (2)

- Programmierer muss sich um Auswertungsreihenfolge und Auswertegrad kümmern
- Primitive `par` und `pseq` in der Praxis schlecht brauchbar
- Lösungsmöglichkeit: `Control.Parallel.Strategies`
  Bietet Strategien zur Koordination der parallelen Berechnung
- Trennung der Strategie von der eigentlichen Berechnung
- Strategie legt Auswertegrad und -reihenfolge fest
- Abhängigkeiten paralleler Berechnungen werden durch Monade explizit ausgedrückt

Eval-Monade

- Modul `Control.Parallel.Strategies` definiert
  die Monade `Eval` zur Erzeugung von Sparks:

```
runEval :: Eval a -> a
rpar :: a -> Eval a -- erzeugt einen Spark
rseq :: a -> Eval a -- wartet auf das Ergebnis
```

- Komponierte Aktionen `Eval`-Monade drücken
  Abhängigkeiten zwischen parallelen Berechnungen aus.

  Z.B. erzwingt `m >>= f` die Berechnung von `m` vor `f`

- Die Monade ist „rein“, hat also keine Seiteneffekte.

Beispiel

```
import Control.Parallel
import Control.Parallel.Strategies

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

main = do
  let myfib = fib
  let s = runEval $ do
            x <- rpar $ myfib 40
            y <- rseq $ myfib 38
            return $ (x+y)
  print s
```

Beispiel: Listenerzeugung

```
import Control.Parallel
import Control.Parallel.Strategies

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

genlist :: Integer -> Integer -> Eval [[Integer]]
genlist _ 0 = return []
genlist x n = do f <- rpar $ fib x
                 let h = [f]
                 t <- genlist x (n-1)
                 return $ h : t
main = do
  let l = runEval $ genlist 38 6
  mapM_ print l
```


Implementierung der Eval-Monade

Sehr einfach:

```
data Eval a = Done a

instance Monad Eval where
  -- return :: a -> Eval a
  return x = Done x
  -- (>>=) :: Eval a -> (a -> Eval b) -> Eval b
  Done x >>= k = k x

runEval :: Eval a -> a
runEval (Done a) = a

rpar :: a -> Eval a
rpar x = x `par` return x

rseq :: a -> Eval a
rseq x = x `pseq` return x
```


Auswertungsstrategien

```
type Strategy a = a -> Eval a
```

Anwendung einer Auswertungsstrategie erfolgt mit `using`:

```
using :: a -> Strategy a -> a
using x  s = runEval (s x)
```

Z.B. kann damit ein sequentieller Haskell-Ausdruck einfach parallelisiert werden:

```
foo1 x y = someexpr
```

wird zu:

```
foo1 x y = someexpr `using` someParallelStrategy
```


Auswertungsstrategien (2)

Einfache Strategien:

```
-- führt keine Auswertung durch
r0 :: Strategy a
r0 x = Done x

-- Parallele Auswertung
rpar :: Strategy a
rpar x = x `par` Done x

-- Auswertung zur WHNF
rseq :: Strategy a
rseq x = x `pseq` Done x

-- komplette Auswertung
rdeepseq :: ... (später)
```

Auswertungsstrategien (3)

Eigene Strategien, Auswertung eines Paares:

```
evalPair :: Strategy a -> Strategy b -> Strategy (a,b)
evalPair sa sb (a,b) = do a' <- sa a
                          b' <- sb b
                          return (a',b')
parEvalPair :: Strategy (a,b)
parEvalPair = evalPair rpar rpar
main = do
  (inp1:inp2:_) <- getArgs
  let x = fib (read inp1)
  let y = fib (read inp2)
  let p = (x, y) `using` parEvalPair
  print $ fst p + snd p  -- p muss verwendet werden!
```

Alternativ mit `Applicative`:

```
evalTuple2 :: Strategy a -> Strategy b -> Strategy (a,b)
evalTuple2 sa sb (a,b) = (,) <$> sa a <*> sb b
```


Auswertungsstrategien (4)

Komposition zweier Strategien:

```
dot :: Strategy a -> Strategy a -> Strategy a
s2 `dot` s1 = s2 . runEval . s1
```

Beispiel:

```
evalList2 (x:y:ys)      =
  do y' <- rseq y
     return (x:y':ys)
evalList2 xs            =
  return xs

evalList3 (x:y:z:zs)    =
  do z' <- rseq z
     return (x:y:z':zs)
evalList3 xs            =
  return xs
```

```
*Main> let xs = [(1+1),(2+2),(3+3),(4+4)]::[Int]
*Main> take 1 xs
[2]
*Main> :sprint xs
xs = [2,_,_,_]
*Main> take 1 (xs `using` evalList3)
[2]
*Main> :sprint xs
xs = [2,_,6,_]
*Main> take 1 (xs `using` evalList2)
[2]
*Main> :sprint xs
xs = [2,4,6,_]
*Main> let xs = [(1+1),(2+2),(3+3),(4+4)]::[Int]
*Main> take 1 (xs `using` evalList2 `dot` evalList3)
[2]
*Main> :sprint xs
xs = [2,4,6,_]
```


Auswertungsstrategien (5)

Parallele Anwendung auf Paare:

```
parPair' :: Strategy a -> Strategy b -> Strategy (a,b)
parPair' strat1 strat2 = evalPair (rpar `dot` strat1) (rpar `dot` strat2)
```

Sequentielle Anwendung auf Listen: wendet gegebene Strategie auf alle Listenelemente an

```
evalList :: Strategy a -> Strategy [a]
     --  :: (a -> Eval a) -> ([a] -> Eval [a])
evalList s [] = return []
evalList s (x:xs) = (:) <$> s x <*> evalList s xs
```

```
*Main> let ys = [(1+1),(2+2),(3+3),(4+4)]::[Int]
*Main> length (ys `using` (evalList r0))
4
*Main> :sprint ys
ys = [_,_,_,_]
*Main> length (ys `using` (evalList rseq))
4
*Main> :sprint ys
ys = [2,4,6,8]
```


Auswertungsstrategien (6)

Listenelemente parallel auswerten:

```
parList :: Strategy a -> Strategy [a]
parList s = evalList (rpar `dot` s)
```

Ein Beispielaufruf dazu ist:

```
main = do
  (inp1:_) <- getArgs
  let ys = [fib (read inp1),fib (read inp1)]
  let ys' = ys `using` (parList rseq)
  print ys'
```

In `ys'` werden alle Listenelemente parallel bis zur WHNF ausgewertet.

Paralleles `map`:

```
parMap :: (a -> b) -> [a] -> [b]
parMap f xs = map f xs `using` parList rseq
```

NFData

Typklasse für Auswertung zur Normalform

```
class NFData a where
  rnf :: a -> ()
  rnf x = x `seq` ()
```

deepseq analog zu seq

```
deepseq :: NFData a => a -> b -> b
deepseq a b = rnf a `seq` b
```

rdeepseq: Strategie zur NF-Auswertung

```
rdeepseq :: NFData a => Strategy a
rdeepseq x = x `deepseq` Done x
```


NFData (2)

Eigene Instanzen von `NFData` sind leicht zu definieren:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
instance NFData a => NFData (Tree a) where
    rnf Leaf = ()
    rnf (Node l a r) = rnf l `seq` rnf a `seq` rnf r
```

Eine generische `NFData`-Instanz für Listen ist:

```
instance NFData a => NFData [a] where
  rnf [] = ()
  rnf (x:xs) = rnf x `seq` rnf xs
```

Beispiel:

```
*Main> let ys = [replicate 2 (1+1), replicate 3 (2*2), replicate 4 (3-3)]::[[Int]]
*Main> length (ys `using` rdeepseq)
3
*Main> :sprint ys
ys = [[2,2],[4,4,4],[0,0,0,0]]
```


Zusammenfassung GpH

- Das zugehörige Paket ist `parallel`
- Parallele Auswertung von Teilausdrücken
- Verwaltung durch RTS mit Sparks
- Kann-Parallelismus
- Auswertungstrategie und Auswertegrad mit Strategien
- Parallelisierung ohne viele Codeänderungen

Par-Monade

Alternative zu GpH, Paket monad-par

Par-Monade

- Strikte Auswertung, weniger Nachdenken über verzögerte Auswertung
- Monade kümmert sich um die deterministische Berechnung
- Deadlocks sind möglich!
- Overhead ist teurer als bei GpH-Sparks
- Kein IO innerhalb der Par-Monade um Determinismus zu garantieren
- Variante (betrachten wir nicht): `Control.Monad.Par.IO`

Explizite Synchronisation in der Par-Monade

`IVar`-Referenzen

```
new :: Par (IVar a)
-- erzeugt eine Referenz auf leeren Speicherplatz

put :: NFData a => IVar a -> a -> Par ()
-- beschreibt Speicherplatz (nur 1x möglich)

get :: IVar a -> Par a
-- liest Speicherplatz, blockiert wenn leer
```

- Deadlock durch zyklische Referenzen möglich
- IVars dürfen nicht zwischen verschiedenen Par-Monaden herumgereicht werden

Programmieren mit der Par-Monade

Grobe Struktur:

```
example :: a -> Par (b,c)
example x = do
    vb <- new  -- vb :: IVar b
    vc <- new  -- vc :: IVar c
    --  Parallele Berechnungen werden nun gestartet
    --  und befüllen vb and vc mit Aufruf an put
    rb <- get vb    -- rb :: b
    rc <- get vc    -- rc :: c
    return (rb,rc)
```

- `new`-Befehle erzeugen leere Speicherplätze,
- `vb` und `vc` werden an die parallelen Berechnungen übergeben
- Die Aufrufe `put vb somevalue` und `put vc othervalue` folgen dort irgendwann
- Die `get`-Befehle synchronisieren

Fork

- Mit `fork :: Par () -> Par ()` wird eine übergebene Berechnung parallel zum aktuellen Thread gestartet.
- Parallele Berechnungen haben kein Ergebnis.
- Ergebnisse müssen als Seiteneffekte in der `Par`-Monade übergeben werden
- durch Beschreiben von `IVar`-Variablen

Beispiel:

```
example :: a -> Par (b,c)
example x = do
    vb <- new                   -- vb :: IVar b
    vc <- new                   -- vc :: IVar c
    fork $ put vb $ taskB x     -- taskB :: a -> b
    fork $ put vc $ taskC x     -- taskC :: a -> c
    rb <- get vb                -- rb == taskB x
    rc <- get vc                -- rc == taskC x
    return (rb,rc)
```

Fork: Beispiele

```
example2 :: Par d
example2 = do
  va <- new; vb <- new; vc <- new; vd <- new
  fork $ put va taskA                          -- A
  fork $ do ra <- get va; put vb $ taskB ra    -- B
  fork $ do ra <- get va; put vc $ taskC ra    -- C
  fork $ do rb <- get vb
            rc <- get vc
            put vd $ taskD rb rc               -- D
  get vd
```

- implizite Abhängigkeiten: `taskB` und `taskC` benötigen Ergebnis von `taskA` und `taskD` benötigt Ergebnisse von `taskB` und `taskC`.
- Reihenfolge der `fork`-Anweisungen ist dabei im Grunde irrelevant

Beispiel mit Deadlock

```
deadlockExample :: Par (b,c)
deadlockExample = do
    vb <- new
    vc <- new
    fork $ do rc <- get vc; put vb $ task1 rc
    fork $ do rb <- get vb; put vc $ task2 rb
    get vb
    get vc
    return (vb,vc)
```


Operationen der Par-Monade

```
runPar   :: Par a  ->    a
runParIO :: Par a  -> IO a
fork     :: Par () -> Par ()
new      ::                          Par (IVar a)
get      ::                IVar a -> Par a
put      :: NFData a => IVar a -> a -> Par ()
put_     ::             IVar a -> a -> Par ()
```

- `runPar` führt die Monade aus
- `fork` startet eine parallele Auswertung,
- `new` erzeugt eine `IVar`,
- `put` erzwingt die volle Auswertung seines Argumentes und füllt die `IVar`,
- `put_` wertet nur bis zur WHNF aus und füllt die `IVar`, und
- `get` wartet bis der Wert verfügbar ist.

Spawn

`spawn :: NFData a => Par a -> Par (IVar a)`

- ist ähnlich zu `fork`
- liefert eine `IVar` zurück für das Ergebnis

Implementierung:

```
spawn p = do
   r <- new
   fork (p >>= put r)
   return r
```

Beispiel:

```
parMap :: NFData b => (a ->     b) -> [a] -> Par [b]
parMap f xs = do ibs <- mapM (spawn . return . f) xs
                 mapM get ibs
```

Zusammenfassung Par-Monade

- `Par`-Monade zur Beschleunigung der Berechnung durch paralleles Auswerten
- Berechnung mit der `Par`-Monade ist deterministisch
- IO-Operationen sind innerhalb `Par`-Monade nicht erlaubt
- Interner Verwaltungsaufwand größer als bei GpH
- Nur größere Berechnungen parallel ausführen
- Parallele Einheiten werden immer ausgewertet, sollten daher benötigt werden
- `Par`-Monade kann jederzeit verwendet werden
- Parallelität nur innerhalb eines `runPar`
- Mehrere `runPar` werden untereinander immer sequentiell ausgewertet

Ausnahmen

Exceptions, Errors,...

Ausnahmen

Fehlschlagen von Berechnung kann mit `Maybe a` und `Either a b` behandelt werden

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

- Prinzipiell ist dies die beste Methode, um mit Ausnahmen umzugehen
- Nachteilig: Behandlung einer Ausnahme ist manchmal nur an einer ganz anderen Stelle im Programm möglich und gesamter Code muss auf den `Maybe`-Typ umgestellt werden
- Abhilfe: Monadisch mit Monaden `Maybe` und `Either a` programmieren (implizites Herumreichen des Fehlers)

Ausnahmen (2)

Speziell zugeschnittene Monade:

```
data Ausnahme = Bauchschmerzen | Kopfweh | Zahnschmerzen Int
type Gesundheit a = Either Ausnahme a

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
```

Ausnahmen (3)

Ausnahmen, die man nicht am Typ erkennt:

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

Insbesondere:

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

Vorteil: `error` kann überall verwendet werden

Ausnahmen im GHC

Klasse `Exception` im Modul `Control.Exception`

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

- `throw` wirft Instanz von `Exception`:

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

- Instanzen von `Exception` bilden Hierarchie, an dessen Wurzel der Typ `SomeException` steht.

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'
```

Aufrufe:

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


Beispiel aus der Standarbibliothek...

```
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)
```

Ausnahmen können nur innerhalb der 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!

Beispiele

```
> 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
```

Nicht ratsam: Alle möglichen Ausnahmen zu fangen

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

Nur Ausnahmen fangen, welche man sinnvoll behandeln kann.

Ausnahme (Exception) vs. Fehler (Error)

- Jede ungefangene Ausnahme ist ein Fehler.
- Ein Fehler führt immer zum Abbruch des Programms
- Exception: Ausnahme, welche speziell behandelt werden muss
- Error: Programmierfehler, Programmierer hat nicht alle Fälle korrekt durchdacht, auch Endlosschleifen, etc.
- Z.B. Datei soll geöffnet werden, aber existiert nicht Kann Ausnahme oder Fehler sein, je nach Behandlung

Ausnahmebehandlung

```
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` reichen eine Ausnahme immer nach außen weiter.

Ausnahmen: Zusammenfassung

- Ausnahmen können überall geworfen werden
- Ausnahmen können nur in der IO-Monade abgefangen werden.
- Ausnahme-Typen bilden eine Hierarchie, welche um benutzerdefinierbare Ausnahmen erweitert werden können.
- Ausnahmen nur gezielt abfangen
- Ausnahme-Behandlung sollte man eher sparsam einsetzen, da dies schnell zu undurchsichtigem Code führt.

Nebenläufigkeit

Concurrent Haskell

Concurrent Haskell

- Explizite Nebenläufigkeit in der IO-Monade im Modul `Control.Concurrent`
- Bibliotheksfunktionen zum Erzeugen und Kontrolle von IO-Threads
- IO-Thread wird nebenläufig zum Hauptthread abgearbeitet.
- auch wenn nur ein Prozessorkern verfügbar ist

```
myThreadId :: IO ThreadId
-- eigene ThreadId

forkIO     :: IO () -> IO ThreadId
-- nebenläufige Berechnung erzeugen

killThread :: ThreadId -> IO ()
-- nebenläufigen Thread töten (unsicher!)
```


Beispiel

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

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

Ausgabe, z.B.

```
BBBBABABABABABABABABABABABABABABABABABAABABABABA...
```


- Nebenläufig erzeugte Threads werden beendet, sobald der Haupthread endet

```
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."
```

- Programm endet zu früh
- Möglichkeit zur Synchronisation von Threads fehlt noch!

MVars

- `MVar a` ist ein veränderlicher Speicherplatz, der leer oder gefüllt sein kann.
- Zugriff in der IO-Monade.
- Grundlegende Operationen sind

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

  Dabei gilt:

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

- Wartende Threads (die auf das Befüllen oder Leeren einer MVar warten) werden dabei durch eine FIFO-Warteschlange an der MVar aufgereit, wodurch Fairness garantiert wird.

Beispiel mit 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."
          syncA <- newEmptyMVar
          syncB <- newEmptyMVar
          syncC <- newEmptyMVar
          forkIO $ showFib "A" 42 >> putMVar syncA ()
          forkIO $ showFib "B" 38 >> putMVar syncB ()
          forkIO $ showFib "C" 40 >> putMVar syncC ()
          mapM_ takeMVar [syncA, syncB, syncC]
          putStrLn "Done."
```


Weitere Beispiele

Exklusiver Zugriff, schütze kritischen Abschnitt durch `takeMVar` und `putMVar`.

Beispiel: Atomarer Zähler

```
type Counter = MVar Integer
newCounter :: IO (Counter)
newCounter i = newMVar i

increaseCounter counter = do
 r <- takeMVar counter
 putMVar counter (r+1)
```


Weitere Beispiele (2)

Deadlocks sind möglich:

```
import Control.Concurrent

main = do
   a <- newEmptyMVar
   b <- newEmptyMVar
   forkIO (takeMVar a >>= putMVar b)
   takeMVar b >>= putMVar a
```

Die Ausführung führt in diesem Fall zum Fehler:
`thread blocked indefinitely in an MVar operation`

Weitere Beispiele (3)

Nichtdeterministisches Mischen:

```
mergeByMVar :: [a] -> [a] -> IO [a]
mergeByMVar xs ys =
 do
  mvar <- newEmptyMVar
  forkIO (mapM_ (putMVar mvar) xs)
  forkIO (mapM_ (putMVar mvar) ys)
  (mapM (\_ -> takeMVar mvar) (xs++ys))
```

Verallgemeinert: Erzeuger-Verbraucher-Probleme mit MVar lösbar Nachteil: Puffergröße ist nur 1, schlecht, wenn die Erzeuger z.B. unterschiedlich schnell sind.

Nicht-strikte Auswertung von putMVar

```
main = do i1:i2:_ <- getArgs
          result1 <- newEmptyMVar
          result2 <- newEmptyMVar
          forkIO $ putMVar result1 (fib (read i1))
          forkIO $ putMVar result2 (fib (read i2))
          r1 <- takeMVar result1
          r2 <- takeMVar result2
          print $ r1 + r2
```

Keine Parallelisierung, da putMVar nicht-strikt im 2.Argument.
Abhilfe:

```
main = do i1:i2:_ <- getArgs
          result1 <- newEmptyMVar
          result2 <- newEmptyMVar
          forkIO $ putMVar result1 $! (fib (read i1))
          forkIO $ putMVar result2 $! (fib (read i2))
          r1 <- takeMVar result1
          r2 <- takeMVar result2
          print $ r1 + r2
```


Unbegrenzte Puffer

- Man kann aus MVars größere veränderbare Strukturen bauen.
- Z.B. Kommunikationskanäle mit unbegrenzten Puffer (FIFO Warteschlangen)
- Diese gibt es aber auch schon fertig in `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 Leseversuch, bis der Channel wieder gefüllt wird.

Weitere Operationen auf MVars

Blockierend:

- `readMVar  :: MVar a -> IO a` zum Lesen
- `swapMVar  :: MVar a -> a -> IO a` zum Austauschen des Werts und
- `modifyMVar_:: MVar a -> (a -> IO a) -> IO ()` zum Anwenden einer Funktion.

Nicht-blockierend:

- `newMVar :: a -> IO (MVar a)` zum Erzeugen einer gefüllten MVar,
- `isEmptyMVar  :: MVar a -> IO Bool` zum Testen, ob eine MVar leer ist
- `tryTakeMVar  :: MVar a -> IO (Maybe a)`
- `tryPutMVar  :: MVar a -> a -> IO Bool`

Asynchrone Ausnahmen

- Manchmal ist es notwendig, laufende Threads zu unterbrechen
- z.B. Benutzer klickt „Abbrechen“
- z.B. Timeout für eine IO-Operationen

Lösungen:

Polling: Thread fragt regelmässig eine MVar ab, ob eine Unterbrechung vorliegt.

Andere Möglichkeit: asynchrone Ausnahmen

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

wirft eine Exception in einem anderen Thread.

Async

Modul `Control.Concurrent.Async` stellt einfache Schnittstelle als Hilfe bereit

```
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 (2)

- `async` ist Ersatz für `forkIO`, `ThreadId` und Ergebnis oder Ausnahme in Rückgabe gesammelt.
- `cancel` analog zu `killThread`
- `waitCatch` wartet, bis der abgezweigte Async-Thread beendet ist
- `wait` wartet auf das Ende und Ergebnis eines Threads.
  Ausnahme wird durch erneutes Werfen einfach nach aussen weitergereicht.

Beispiel

Webseiten downloaden, mit `'q'` abbrechen

```
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)
```

Probleme bei asynchronen Ausnahmen

```
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>
```

Wird die Funktion durch einen anderen Thread unterbrochen zwischen <1> und <2> oder auch zwischen <2> und <4>, so bleibt die MVar leer.

Dadurch wird ein inkonsistenter Zustand erzeugt, der z.B. zu Deadlocks in anderen Threads führen kann, wenn „mv nie leer“ als Invariante angenommen wurde.

Masking

Primitive zum Abschirmen vor Ausnahmen:

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

unterdrückt Ausnahmen temporär.

Beispiel:

```
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
```

Mit `restore` werden Ausnahmen für `(fkt a)` temporär erlaubt!
Blockierte Aktionen (z.B. `takeMVar` und `putMVar`) können trotzdem unterbrochen werden!

Masking (2)

Masking wird an verschiedenen Stellen eingesetzt, z.B. auch in der bereits 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
```

stellt sicher, dass die Aufräumaktionen in jedem Fall durchgeführt werden.

async

`async` war fehlerhaft, nun richtig

```
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)
```

Stellt sicher, dass die MVar gesetzt wird:

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

`forkIO` übernimmt immer die aktuelle Maskierung!

forkFinally

Da dieses Muster häufiger auftritt, gibt es `forkFinally`:

```
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)
```


Software Transactional Memory

Grundlagen

Problem bei sperrenbasierter Programmierung:

```
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)
```

- Implementiere überweise:
  ```
  transfer :: Int -> Account -> Account -> IO ()
  transfer n (Account from) (Account to) = do
    bal_from <- takeMVar from
    bal_to   <- takeMVar to
    putMVar from (bal_from - n)
    putMVar to   (bal_to   + n)
  ```
- Sperren müssen überdacht werden
- nicht korrekt, da Deadlocks auftreten können!

Probleme der Lock-basierten Programmierung

- Bei zu wenigen Locks drohen Race Conditions und verletzte Invarianten
- Bei zu vielen Locks wird das Programm sequentialisiert oder sie verursachen sogar Deadlocks
- Die Reihenfolge in der Locks gesetzt werden ist unklar, bzw. muss global festgelegt werden
- Die Platzierung der Locks ist oft unklar
- Die Ausnahmebehandlung kann schwierig sein, da alle Locks in einen konsistententen Zustand gebracht werden müssen

Transactional Memory

- Abhilfe ist die Idee des Transactional Memory
- Threads greifen auf gemeinsamem Speicher durch Transaktionen zu
- Transaktion ist ein Bündel von Speicherzugriffen eines Threads, welche atomar ausgeführt werden
- Zwischenzustände werden für andere Threads unsichtbar
- Transaktionen werden überwacht nebenläufig ausgeführt; bei einem Konflikt von Speicherzugriffen wird die Transaktion abgebrochen, alle bisherigen Effekte rückgängig gemacht und später wiederholt

STM Haskell

- Software Transactional Memory (STM)
- Rollback abgebrochener Transaktionen ist in der funktionaler Welt recht einfach
- Modul `Control.Concurrent.STM`
- Monade `STM` bündelt Blöcke von Zugriffen auf gemeinsame Variablen (`TVar`) zu Transaktionen zusammen
- Wie in jeder Monade können diese Blöcke miteinander kombiniert werden
- `STM`-Transaktion könnenen innerhalb der `IO`-Monade scheinbar atomar und ohne Blockierung (lock-free) ausgeführt werden
- kein IO in der `STM`-Monade

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
```

Atomares Ausführen einer Transaktion in der `IO`-Monade wird

durch

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

durchgeführt.

Ausführung

- Intern werden Lese- und Schreibzugriffe auf TVars in einem Log festgehalten
- Wenn Transaktion beendet, wird Log geprüft
- Bei Konflikten wird ein „roll-back“ durchgeführt:
  Log verwerfen, Transaktion neu starten
- Bei keinem Konflikt werden die Schreibzugriffe auf dem echten Speicher ausgeführt (mit vorübergehendem Sperren der TVars)

Beispiel

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

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


Beispiel: Überweisen

```
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)
```


Blockieren

- Auch mit STM kann man blockieren
- `retry :: STM a` löst ein explizites roll-back aus
  Der Thread wird angehalten, bis sich mindestens eine der bis dahin *gelesenen* `TVar` verändert hat

Z.B. 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)
```


orElse

Alternative Auswahl mit

`orElse :: STM a -> STM a -> STM a`

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.

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
```

Beispiel

Implementierung von MVars mithilfe von STM-Haskell:

```
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
```


Ausnahmebehandlung in STM

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

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

- Funktionieren wie in der `IO`-Monade, außer, dass auch bei `catchSTM` alle Seiteneffekte des ersten Arguments verworfen werden, bevor der Handler ausgeführt wird

Nachteile und Probleme

- `STM`-Aktionen sollten nicht zu groß werden, da ggf. alles wiederholt werden muss
- 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
- Allgemein ist STM-Code langsamer als `MVar`-Code
  z.B. `readTVar` lineare Laufzeit in der Anzahl der Zugriffe, da bei jedem Zugriff das `TVar`-Transaktionen-Log der Aktion geprüft werden muss