Einführung in die Funktionale Programmierung

Striktheit & Parallelität

Steffen Jost

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

10. Juli 2013

Lazy Evaluation

Teilausdrücke werden als **Thunk** gespeichert, und erst bei Bedarf zu einem **Wert** ausgewertet. Im Speicher wird der Thunk dann durch seinen Wert ersetzt, was mehrfache Auswertung verhindert.

Vorteile

- Effizient: Maximal einmal auswerten, wenn überhaupt
- Verwendung potentiell unendlicher Datenstrukturen
- Erlaubt gute Modularisierung durch Trennung von Daten und Kontrollfluss

Nachteile

- Keine Sequentialität der Auswertung (erst das, dann das)
- Höherer Speicherverbrauch zum Merken der Verweise und Teilausdrücke

Lazy Evaluation

Lazy Evaluation ermöglicht Trennung von Daten und Kontrollfluss, auch ohne den Einsatz von Zirkularität:

Beispiel

```
factors :: Integral a => a -> [a]
factors n = filter (\m -> n `mod` m == 0) [2 .. (n - 1)]

isPrime :: Integral a => a -> Bool
isPrime n = n > 1 && null (factors n)
```

`factors` berechnet alle Faktoren einer Zahl.

Die Berechnung von `isPrime` bricht sofort ab, so bald der erste Faktor gefunden wurde, trotz Verwendung von `factors` werden also nicht alle Faktoren berechnet!

Strikte Sprachen

In anderen Programmiersprachen ist die Idee nicht unbekannt:
z.B. bieten C und Java faule Varianten von logischen Operatoren:
Der Java-UND-Operator `&` wertet immer beide Argumente aus;
aber `&&` wertet das zweite Argument nicht aus, wenn das erste bereits `False` ergibt. ⇒ short-circuit

Programmiersprachen, in denen im Gegensatz zur verzögerten Auswertung alle Ausdrücke sofort ausgewertet werden, nennt man **strikt** engl. eager. Fast alle imperativen Sprachen sind strikt.

Faule Auswertung kann in strikten Sprachen simuliert werden, in dem man Ausdrücke zu Funktionen mit Scheinargumenten macht:

```
foo x = let e' = (\_ -> e) -- keine Auswertung von e
        in ... e' () ...   -- Auswertung von e
```

Speicherverbrauch Lazy Evaluation

Beispiel

```
sumWithL :: [Int] -> Int -> Int
sumWithL   []  acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h
```

```
sumWithL [2,3,4]    1              ⇝
sumWithL   [3,4]   (1+2)           ⇝
sumWithL     [4]  ((1+2)+3)        ⇝
sumWithL      [] (((1+2)+3)+4)     ⇝
                  ((1+2)+3)+4      ⇝
                    ((3)+3)+4      ⇝ (6)+4 ⇝ 10
```

Obwohl sumWithL endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹

Speicherverbrauch Lazy Evaluation

Beispiel

```
sumWithL :: [Int] -> Int -> Int
sumWithL   []  acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h
```

```
sumWithL [2,3,4]    1          ⇝
sumWithL   [3,4]   (1+2)       ⇝
sumWithL     [4]  ((1+2)+3)    ⇝
sumWithL      [] (((1+2)+3)+4) ⇝
                  ((1+2)+3)+4  ⇝
                    ((3)+3)+4  ⇝ (6)+4 ⇝ 10
```

Obwohl sumWithL endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹

Speicherverbrauch Lazy Evaluation

Beispiel

```
sumWithL :: [Int] -> Int -> Int
sumWithL   []  acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h
```

```
sumWithL [2,3,4]    1          ⇝
sumWithL   [3,4]   (1+2)       ⇝
sumWithL     [4]  ((1+2)+3)    ⇝
sumWithL      [] (((1+2)+3)+4) ⇝
                 ((1+2)+3)+4   ⇝
                   ((3)+3)+4   ⇝ (6)+4 ⇝ 10
```

Obwohl sumWithL endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹

Speicherverbrauch Lazy Evaluation

Beispiel

```
sumWithL :: [Int] -> Int -> Int
sumWithL   []  acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h
```

```
sumWithL [2,3,4]    1          ⇝
sumWithL   [3,4]   (1+2)       ⇝
sumWithL     [4]  ((1+2)+3)    ⇝
sumWithL      [] (((1+2)+3)+4) ⇝
                 ((1+2)+3)+4   ⇝
                   ((3)+3)+4   ⇝ (6)+4 ⇝ 10
```

Obwohl sumWithL endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹

Speicherverbrauch Lazy Evaluation

Beispiel

```
sumWithL :: [Int] -> Int -> Int
sumWithL   []  acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h
```

```
sumWithL [2,3,4]    1          ⇝
sumWithL   [3,4]   (1+2)       ⇝
sumWithL     [4]  ((1+2)+3)    ⇝
sumWithL      [] (((1+2)+3)+4) ⇝
                  ((1+2)+3)+4  ⇝
                    ((3)+3)+4  ⇝ (6)+4 ⇝ 10
```

Obwohl sumWithL endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹

Speicherverbrauch Lazy Evaluation

Beispiel

```
sumWithL :: [Int] -> Int -> Int
sumWithL   []  acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h
```

```
sumWithL [2,3,4]    1           ⇝
sumWithL   [3,4]   (1+2)        ⇝
sumWithL     [4]  ((1+2)+3)     ⇝
sumWithL      [] (((1+2)+3)+4)  ⇝
                  ((1+2)+3)+4   ⇝
                    ((3)+3)+4   ⇝ (6)+4 ⇝ 10
```

Obwohl sumWithL endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹

Speicherverbrauch Lazy Evaluation

Beispiel

```
sumWithL :: [Int] -> Int -> Int
sumWithL   []  acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h

sumWithL [2,3,4]    1           ⇝
sumWithL   [3,4]   (1+2)        ⇝
sumWithL     [4]  ((1+2)+3)     ⇝
sumWithL      [] (((1+2)+3)+4)  ⇝
                  ((1+2)+3)+4   ⇝
                    ((3)+3)+4   ⇝ (6)+4 ⇝ 10
```

Obwohl `sumWithL` endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹

Speicherverbrauch Lazy Evaluation

Beispiel

```
sumWithL :: [Int] -> Int -> Int
sumWithL   []  acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h
```

```
sumWithL [2,3,4]    1           ⇝
sumWithL   [3,4]   (1+2)        ⇝
sumWithL     [4]  ((1+2)+3)     ⇝
sumWithL      [] (((1+2)+3)+4)  ⇝
                  ((1+2)+3)+4   ⇝
                    ((3)+3)+4   ⇝ (6)+4 ⇝ 10
```

Obwohl `sumWithL` endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹

Speicherverbrauch Lazy Evaluation

Beispiel

```
sumWithL :: [Int] -> Int -> Int
sumWithL   []  acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h

sumWithL [2,3,4]    1           ⇝
sumWithL   [3,4]   (1+2)        ⇝
sumWithL     [4]  ((1+2)+3)     ⇝
sumWithL      [] (((1+2)+3)+4)  ⇝
                  ((1+2)+3)+4   ⇝
                    ((3)+3)+4   ⇝ (6)+4 ⇝ 10
```

Obwohl `sumWithL` endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹

Speicherverbrauch Lazy Evaluation

Beispiel

```
sumWithL :: [Int] -> Int -> Int
sumWithL   []  acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h

sumWithL [2,3,4]    1           ⇝
sumWithL   [3,4]   (1+2)        ⇝
sumWithL     [4]  ((1+2)+3)     ⇝
sumWithL      [] (((1+2)+3)+4)  ⇝
                  ((1+2)+3)+4   ⇝
                    ((3)+3)+4   ⇝ (6)+4 ⇝ 10
```

Obwohl `sumWithL` endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹

Striktheit erzwingen

Haskell bietet daher einen Mechanismus, welcher die **strikte Auswertung** erzwingt:

```
($!) :: (a -> b) -> a -> b
```

Der Ausdruck `f $! x` erzwingt die Auswertung von `x` vor der Anwendung von `f`.

Beispiel:

```
sumWithS :: [Int] -> Int -> Int
sumWithS   []  acc = acc
sumWithS (h:t) acc = sumWithS t $! acc+h

sumWithS [2,3,4]     1   ⇝
sumWithS   [3,4] $! (1+2) ⇝ sumWithS [3,4] 3
sumWithS     [4] $! (3+3) ⇝ sumWithS   [4] 6
sumWithS      [] $! (6+4) ⇝ sumWithS   [] 10 ⇝ 10
```

Striktheit erzwingen

Haskell bietet daher einen Mechanismus, welcher die **strikte Auswertung** erzwingt:

```
($!) :: (a -> b) -> a -> b
```

Der Ausdruck `f $! x` erzwingt die Auswertung von `x` vor der Anwendung von `f`.

Beispiel:

```
sumWithS :: [Int] -> Int -> Int
sumWithS   []  acc = acc
sumWithS (h:t) acc = sumWithS t $! acc+h
```

```
sumWithS [2,3,4]     1    ⇝
sumWithS   [3,4] $! (1+2) ⇝ sumWithS [3,4] 3
sumWithS     [4] $! (3+3) ⇝ sumWithS   [4] 6
sumWithS      [] $! (6+4) ⇝ sumWithS   [] 10 ⇝ 10
```

Striktheit erzwingen

Haskell bietet daher einen Mechanismus, welcher die **strikte Auswertung** erzwingt:

```
($!) :: (a -> b) -> a -> b
```

Der Ausdruck `f $! x` erzwingt die Auswertung von `x` vor der Anwendung von `f`.

Beispiel:

```
sumWithS :: [Int] -> Int -> Int
sumWithS   []  acc = acc
sumWithS (h:t) acc = sumWithS t $! acc+h

sumWithS [2,3,4]     1    ⇝
sumWithS   [3,4] $! (1+2) ⇝ sumWithS [3,4] 3
sumWithS     [4] $! (3+3) ⇝ sumWithS   [4] 6
sumWithS      [] $! (6+4) ⇝ sumWithS    [] 10 ⇝ 10
```

Striktheit erzwingen

Haskell bietet daher einen Mechanismus, welcher die **strikte Auswertung** erzwingt:

```
($!) :: (a -> b) -> a -> b
```

Der Ausdruck `f $! x` erzwingt die Auswertung von `x` vor der Anwendung von `f`.

Beispiel:

```
sumWithS :: [Int] -> Int -> Int
sumWithS   []  acc = acc
sumWithS (h:t) acc = sumWithS t $! acc+h

sumWithS [2,3,4]     1    ⇝
sumWithS   [3,4] $! (1+2) ⇝ sumWithS [3,4] 3
sumWithS     [4] $! (3+3) ⇝ sumWithS   [4] 6
sumWithS      [] $! (6+4) ⇝ sumWithS   [] 10 ⇝ 10
```

Striktheit erzwingen

Haskell bietet daher einen Mechanismus, welcher die **strikte Auswertung** erzwingt:

```
($!) :: (a -> b) -> a -> b
```

Der Ausdruck `f $! x` erzwingt die Auswertung von `x` vor der Anwendung von `f`.

Beispiel:

```
sumWithS :: [Int] -> Int -> Int
sumWithS   []  acc = acc
sumWithS (h:t) acc = sumWithS t $! acc+h

sumWithS [2,3,4]     1    ⇝
sumWithS   [3,4] $! (1+2) ⇝ sumWithS [3,4] 3
sumWithS     [4] $! (3+3) ⇝ sumWithS   [4] 6
sumWithS      [] $! (6+4) ⇝ sumWithS   [] 10 ⇝ 10
```

Striktheit erzwingen

Haskell bietet daher einen Mechanismus, welcher die **strikte Auswertung** erzwingt:

```
($!) :: (a -> b) -> a -> b
```

Der Ausdruck `f $! x` erzwingt die Auswertung von `x` vor der Anwendung von `f`.

Beispiel:

```
sumWithS :: [Int] -> Int -> Int
sumWithS   []  acc = acc
sumWithS (h:t) acc = sumWithS t $! acc+h

sumWithS [2,3,4]     1    ⇝
sumWithS   [3,4] $! (1+2) ⇝ sumWithS [3,4] 3
sumWithS     [4] $! (3+3) ⇝ sumWithS   [4] 6
sumWithS      [] $! (6+4) ⇝ sumWithS   [] 10 ⇝ 10
```

Striktheit erzwingen

Haskell bietet daher einen Mechanismus, welcher die **strikte Auswertung** erzwingt:

```
($!) :: (a -> b) -> a -> b
```

Der Ausdruck `f $! x` erzwingt die Auswertung von `x` vor der Anwendung von `f`.

Beispiel:

```
sumWithS :: [Int] -> Int -> Int
sumWithS   []  acc = acc
sumWithS (h:t) acc = sumWithS t $! acc+h

sumWithS [2,3,4]     1    ⇝
sumWithS   [3,4] $! (1+2) ⇝ sumWithS [3,4] 3
sumWithS     [4] $! (3+3) ⇝ sumWithS   [4] 6
sumWithS      [] $! (6+4) ⇝ sumWithS   [] 10 ⇝ 10
```

Striktheit erzwingen

Haskell bietet daher einen Mechanismus, welcher die **strikte Auswertung** erzwingt:

```
($!) :: (a -> b) -> a -> b
```

Der Ausdruck `f $! x` erzwingt die Auswertung von `x` vor der Anwendung von `f`.

Beispiel:

```
sumWithS :: [Int] -> Int -> Int
sumWithS   []  acc = acc
sumWithS (h:t) acc = sumWithS t $! acc+h

sumWithS [2,3,4]     1    ⇝
sumWithS   [3,4] $! (1+2) ⇝ sumWithS [3,4] 3
sumWithS     [4] $! (3+3) ⇝ sumWithS   [4] 6
sumWithS      [] $! (6+4) ⇝ sumWithS   [] 10 ⇝ 10
```

Striktheit erzwingen

`$!` ist definiert durch das Primitiv `seq :: a -> b -> b`

```
($!) :: (a -> b) -> a -> b
f $! x  = x `seq` f x
```

`seq` erzwingt die Auswertung seines ersten Argumentes und liefert danach das zweite Argument zurück.

```
foo x =
  let zwischenwert = bar x
      ergebnis     = goo zwischenwert
  in  seq zwischenwert ergebnis
```

Von einigen Funktionen bietet die Standardbibliothek auch strikte Varianten, z.B. macht `foldl'` das gleiche wie `foldl`, erzwingt jedoch die Auswertung des Akkumulators in jedem Schritt.

Striktheit erzwingen

Das erste Argument von `seq` wird nur soweit ausgewertet, bis dessen äußere Form klar ist, darin kann auf weitere Thunks verwiesen werden:

Int, Bool: werden vollständig ausgewertet

Listen: ausgewertet bis klar ist, ob Liste leer ist oder nicht. Kopf und der Rumpf werden nicht ausgewertet!

Tupel: ausgewertet bis Tupel-Konstruktor fest steht, d.h. Elemente des Tupels werden nicht ausgewertet.

Maybe: ausgewertet bis `Nothing` oder `Just`, das Argument von Just wird noch nicht ausgewertet.

Generell wertet `seq` bis zum äußeren Konstruktor aus. Argumente des Konstruktors werden nicht weiter ausgewertet.

Schwache Kopf-Normalform, engl. Weak Head Normal Form

Demo

Demo `sumWith.hs`:
Wir führen `sumWithL` und `sumWithS` für große Listen mit GHC aus und werden überrascht!

Demo

Demo `sumWith.hs`:
Wir führen `sumWithL` und `sumWithS` für große Listen mit GHC aus und werden überrascht!

- Wir haben gesehen, dass der Kompilier automatisch eine Striktheit-Analyse durchführt, d.h. wir müssen nur selten eingreifen.
- Wenn ein Stack-Overflow eintritt, dann sollte man über Endrekursion und auch über Striktheit nachdenken.

Probleme mit Striktheit

Einfügen von `seq` kann ein funktionierendes Programm zerstören:

```
> foldr  (&&) True (repeat False)
False

> foldr' (&&) True (repeat False)  -- aus Data.Foldable
<interactive>: Heap exhausted;

> let foo x y = x
> foo 42 $  undefined
42

> foo 42 $! undefined
*** Exception: Prelude.undefined
```

Eine Funktion, welche für `undefined` in Argument *n* immer einen Fehler liefern, nennt man auch **strikt im *n*-ten Argument**.

Probleme mit Striktheit

Einfügen von `seq` kann ein funktionierendes Programm zerstören:

```
> foldr  (&&) True (repeat False)
False

> foldr' (&&) True (repeat False)  -- aus Data.Foldable
<interactive>: Heap exhausted;

> let foo x y = x
> foo 42 $  undefined
42

> foo 42 $! undefined
*** Exception: Prelude.undefined
```

Eine Funktion, welche für `undefined` in Argument *n* immer einen Fehler liefern, nennt man auch **strikt im *n*-ten Argument**.

Zusammenfassung

- Faule Auswertung kann zu erhöhtem Speicherverbrauch führen
- Vermeidbar mit expliziten Striktheits-Operatoren
- Einstreuen von `seq` und `$!` kann Programmabbrüche provozieren, insbesondere im Zusammenhang mit automatischen Optimierungen des Kompilers
- Kompiler kann ohnehin Probleme der verzögerten Auswertung automatisch vermeiden

Grundlagen

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

Computer mit mehreren Kernen und Cloud-Architekturen sind Standard und ermöglichen paralleles Rechnen.
Problem: Viele Ansätze für paralleles Rechnen sind sehr "low-level" und schwierig zu handhaben.

Parallele funktionale Sprachen bieten einen hochsprachlichen Ansatz, in dem nur wenige Aspekte der parallelen Berechnung bestimmt werden müssen!

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

Grundlagen

Parallele Berechnung ist schwierig:

Berechnung

korrekter und effizienter Algorithmus zur Berechnung des Gewünschten (wie in sequentieller Berechnung)

Koordination

Sinnvolle Einteilung der Berechnung in unabhängige Einheiten, welche parallel ausgewertet werden können

Beurteilung der Effizienz erfolgt primär durch Vergleich der Beschleunigung relativ zur Berechnung mit einem Prozessor.

Beispiel: Faktor 14 ist gut, wenn anstatt 1 Prozessor 16 verwendet werden, aber schlecht, wenn 128 verwendet werden (dürfen).

Koordinierung der parallelen Ausführung

Partitionierung: Aufspaltung des Programms in unabhängige, parallel berechenbare Teile, **Threads**

- Wie viele Threads?
- Wie viel macht ein einzelner Thread?

Synchronisation
Abhängigkeiten zwischen Threads identifizieren

Kommunikation / Speicher Management
Austausch der Daten zwischen den Threads

Mapping Zuordnung der Threads zu Prozessoren

Scheduling Auswahl lauffähiger Threads auf einem Prozessor

⇒ Explizite Spezifizierung durch den Programmierer sehr aufwändig und auch sehr anfällig für Fehler!

Koordinierung der parallelen Ausführung

Existierende Sprachen unterscheiden sich stark im Grad der Kontrolle dieser Aspekte. Es gibt verschiedene Ansätze, um den Aufwand zur Koordinierung der parallelen Ausführung für den Programmier zu reduzieren z.B. Skeletons

Manche Sprachen verwenden eigene **Koordinierungssprachen** die diese Aspekte kontrollieren

Viele der verwendeten Sprachen bieten lediglich Bibliotheken o.ä. mit denen eine explizite Kontrolle möglich ist

Parallele Funktionale Sprachen

Rein funktionale Programmiersprachen haben keine Seiteneffekte und sind **referentiell transparent**.

Stoy (1977):

> *The only thing that matters about an expression is its value, and any sub-expression can be replaced by any other equal in value. Moreover, the value of an expression is, within certain limits, the same wherever it occurs.*

Insbesondere ist die Auswertungsreihenfolge (nahezu) beliebig.

$\Rightarrow$ Ideal, um verschiedene Programmteile parallel zu berechnen!

Ansätze zur Parallelität in Haskell

- **Glasgow Parallel Haskell**: Hinweise zur Partitionierung werden ins Programm eingestreut. Kontrolle der Parallelität erfolgt automatisch im Laufzeitsystem.
- **Software Transactional Memory:** Bibliothek erlaubt *spekulativ* parallele Berechnungen durchzuführen. Am Ende der Berechnung wird auf mögliche Konflikte mit anderen Berechnungen getestet ("lock-free").
  Erlaubt auch volle Nebenläufigkeit.
- **Par-Monade**: Monade für globales Scheduling und Kommunikation zwischen Threads.
- **Nested Data Parallelism:** Parallelität ist beschränkt auf gleichzeitiges Ausführen einer Operation auf (großen) Datenstrukturen, z.B. mittels Array-comprehensions.

Verwendung mehrerer Kerne in GHC

Anzahl der verwendeten Kerne in GHC einstellen mit

- **Kompiler-Parametern:**
  Für 32 Kerne kompilieren mit

```
ghc prog.hs -threaded -with-rtsopts="-N32"
```

  Alternativ kompilieren mit

```
ghc prog.hs -threaded -rtsopts
```

  und dann Aufrufen mit `./prog +RTS -N32`

  RTS=RunTime System

- **Dynamisch im Programm** mit Modul **Control.Concurrent**

```
getNumCapabilities :: IO Int

setNumCapabilities :: Int -> IO ()
setNumCapabilities 32
```

  Wert kann zur Laufzeit nur erhöht, nie herabgesetzt werden!

Semi-explizite Parallelität: GpH

Glasgow parallel Haskell (GpH) Modul **Control.Parallel**

erweitert Haskell konservative durch zwei Primitive:

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

`x` ‘`par`‘ `e` definiert die parallele Auswertung von `x` und von `e`.

- gibt Hinweis auf parallele Auswertung; nicht erzwungen
- erzeugt **Spark** für `x`; wird ausgewertet, falls Prozessor frei

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

`x` ‘`pseq`‘ `e` definiert sequentielle Auswertung von `x` und von `e`;

- `x` wird zur Weak Head Normal Form ausgewertet
- verbietet dem Kompilier einige Optimierungsmöglichkeiten im Vergleich zum konventionellen `seq` für strikte Auswertung

Beispiel: parfact

Parallele Faktorial Berechnung — Wir wollen *factorial* Funktion parallelisieren, klassisches **divide and conquer**:

```
parfact :: Integer -> Integer
parfact n = parfact' 1 n

parfact' :: Integer -> Integer -> Integer
parfact' m n
  | m == n    = m
  | otherwise = left `par` right `pseq` (left * right)
    where mid   = (m + n) `div` 2
          left  = parfact' m mid
          right = parfact' (mid+1) n
```

- Auswertereihenfolge muss kontrolliert werden: Auswertung Multiplikation könnte erneut Auswertung von `left` verlangen, und nicht parallele Auswertung von `right`
- `pseq` erzwingt Auswertung von `left` und `right` vor

Beispiel: parfact

Verbesserung des Codes durch "thresholding":

```
parfact2 :: Integer -> Integer
parfact2 n = parfact2' 1 n 50

parfact2' :: Integer -> Integer -> Integer -> Integer
parfact2' m n t
  | (n - m) <= t = product [m..n]
  | otherwise = left `par` right `pseq` (left * right)
      where mid = (m + n) `div` 2
            left = parfact2' m mid t
            right = parfact2' (mid + 1) n t
```

- Overhead für Verwaltung der Sparks kostet Zeit, daher besser zu viele "kleine" Sparks vermeiden

Sparks

Ein Spark steht für eine *mögliche* parallele Berechnung. Zur Laufzeit gibt es mehrer Möglichkeiten für einen Spark:

Konvertiert engl. converted

Der Spark wurde ausgerechnet

Abgeschnitten engl. pruned

Spark wurde nicht ausgerechnet

- **Dud**: Spark war schon ein ausgerechneter Wert
- **Fizzled**: Anderer Thread berechnete Wert schneller
- **Garbage Collected**: Keine Referenz zum Spark vorhanden, d.h. Wert wird nicht mehr benötigt

Anzeige der Laufzeit Statistik mit:

```
> ghc Program.hs --make -O2 -threaded -rtsopts
> ./Program +RTS -N3 -s
```

Idealwerweise sollten deutlich mehr Spark konvertiert als abgeschnitten werden!

Beispiel: Quicksort

Naive Version von Quicksort erzeugt nur geringe Parallelität:

```
qsort1 :: Ord a => [a] -> [a]
qsort1 []  = []
qsort1 [x] = [x]
qsort1 (x : xs) = qlo `par`
                  qhi `par`
                  (qlo ++ (x:qhi))
  where qlo = qsort1 [ y | y <- xs, y <  x ]
        qhi = qsort1 [ y | y <- xs, y >= x ]
```

- Kaum parallele Auswertung, da alle Threads sehr schnell WHNF erreichen (erste Cons-Zelle jeder Subliste)
- Wir müssen daher strikte Auswertung des Ergebnis erzwingen

Beispiel: quick-sort

Auswertung der Teillisten muss forciert werden:

```
qsort2 :: Ord a => [a] -> [a]
qsort2 []       = []
qsort2 [x]      = [x]
qsort2 (x : xs) = forcelist qlo `par`
                  forcelist qhi `par`
                  (qlo ++ (x:qhi))
  where qlo = qsort2 [ y | y <- xs, y <  x ]
        qhi = qsort2 [ y | y <- xs, y >= x ]

forcelist :: [a] -> ()
forcelist []     = ()
forcelist (x:xs) = x `seq` forcelist xs
```

Problem: "forcing" auf vielen verschiedenen Datenstrukturen benötigt — und deren Kompositionen listen, listen von listen, ...

Auswertestrategien

Evaluation strategies bieten von Berechnung getrennte Abstraktion der parallelen Koordination Control.Parallel.Strategies

Definiert lediglich Koordination, also Abstraktion über `par` & `pseq`

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

Eine simple Auswertungsfunktion

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

und `return` Operator zum Einführen in die `Eval` Monade:

```
return :: a -> Eval a
return x = Done x
```

Auswertestrategien

Anwendung einer Auswertestrategie erfolgt mit `using`:

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

GpH Beispiel:

```
somefun x y = someexpr `using` somestrat
```

Auswertestrategien mit Auswertungsgrad

Einfache Strategien zur Kontrolle von Auswertegrad, Auswertereihenfolge und Parallelität:

- `r0` führt keine Auswertung durch
- `rseq` führt eine Auswertung zur WHNF durch (default)
- `rdeepseq` führt eine komplette Auswertung durch
- `rpar` führt Auswertung parallel durch

```
r0 :: Strategy a
r0 x = Done x

rseq :: Strategy a
rseq x = x `pseq` Done x

rpar :: Strategy a
rpar x = x `par` Done x
```

Beispiel: parfact mit Strategie

```
parfact3 :: Integer -> Integer
parfact3 n = parfact3' 1 n

parfact3' :: Integer -> Integer -> Integer
parfact3' m n
  | m == n    = m
  | otherwise = (left * right) `using` strategy
    where mid   = (m + n) `div` 2
          left  = parfact3' m mid
          right = parfact3' (mid+1) n
          -- strategy = r0
          strategy result = do
            rpar left
            rpar right
            return result
```

Kontrolle des Auswertegrad

Mit `r0`, `rseq` und `rdeepseq` wird der Auswertegrad eines Ausdruck reguliert.

`rdeepseq` wertet vollständig aus; dies wird über die `rnf` Strategie definiert, welche von der Klasse `NFData` bereitgestellt wird.

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

Für viele Basistypen entspricht `rnf` genau `rwhnf`;
viele Instanzen sind vordefiniert. Control.Parallel.Strategies

Kontrolle des Auswertegrad: rdeepseq

Instanzen für `NFData` werden beispielsweise so definiert:

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

Es gibt auch einen `deepseq` Operator, welcher sein erstes Argument vollständig auswertet:

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

Die `rdeepseq` Auswertestrategie wird oft verwendet:

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

Auswertestrategien kombinieren

Auswertestrategien können auch kombiniert werden:

Komposition

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

Sequentielle Anwendung auf Listen

```
evalList :: Strategy a -> Strategy [a]
evalList s [] = return []
evalList s (x:xs) = do x' <- s x
                       xs' <- evalList s xs
                       return (x':xs')
```

Auswertestrategien kombinieren

Parallele Anwendung auf Listen

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

Parallele Anwendung auf Tupel

```
parTuple2 :: Strategy a ->
             Strategy b -> Strategy (a,b)
parTuple2 strat1 strat2 =
    evalTuple2 (rpar `dot` strat1)
               (rpar `dot` strat2)
```

Semi-explizite Parallelität mit GpH

GpH: Glasgow parallel Haskell Control.Parallel

- Erlaubt parallele Auswertung von Teilausdrücken/Thunks
- Verwaltung der Parallelität erfolgt im Laufzeitsystem
  Overhead ist geringer als in anderen Ansätzen
- Programmierer entscheidet über **Auswertereihenfolge**
- Programmierer entscheidet über **Auswertegrad**
- Kombinierbare Auswertestrategien erfassen
  Auswertereihenfolge und Auswertegrad
- Funktionaler Code mit wenigen Änderungen parallelisierbar!

Autoren: P.W. Trinder, K. Hammond, H-W. Loidl,
Simon L. Peyton Jones.

Explizite Parallelität

`par` kann rein funktionalen Code einfach parallelisieren, aber:

- in vielen großen Anwendung findet man oft zustandsbasierten (monadischen) Code, den man parallel abarbeiten will
- manche Anwendung bestehen ganz natürlich aus einer Menge nebenläufiger Threads

$\Rightarrow$ **Software Transactional Memory**

Nebenläufigkeit bedeutet nicht automatisch die Verwendung mehrer Kerne, sondern strukturiert ein Programm in mehrere asynchron ablaufende Einheiten!

Concurrent Haskell

Concurrent Haskell (Modul: Control.Concurrent) bietet Bibliotheksfunktionen zum Erzeugen und zur Kontrolle von nebenläufigen Berechnungen (IO-Thread).

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

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

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

Beispiel Nebenläufigkeit

```
import Control.Concurrent
x = 35
f = fib
sillyA = putStrLn $ "SillyA " ++ (show $ f (x+9))
sillyB = putStrLn $ "SillyB " ++ (show $ f (x+0))
sillyC = putStrLn $ "SillyC " ++ (show $ f (x-9))

main = do putStrLn "Creating Threads."
          forkIO sillyA
          forkIO sillyB
          forkIO sillyC
          putStrLn "Waiting for completion."
          threadDelay 15000000
          putStrLn "Done."
```

Achtung: Implizite Sychronisation durch gemeinsame Thunks!

Explizite Synchronisation: MVars

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

- `MVar a` ist eine Speicherstelle des Typs a
- Zugriff innerhalb der IO-Monade möglich
- MVar kann leer oder gesetzt sein

Die zwei grundlegenden Operationen sind

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

Semantik:

	MVar Leer	MVar Gesetzt
takeMVar	Blockiert	liefert & leert MVar
putMVar	Setzt MVar	Blockiert

Explizite Synchronisation: MVars

Zahlreiche Operationen im Modul `Control.Concurrent.MVar` verfügbar:

`newEmptyMVar :: IO (MVar a)` erzeugte neue leere MVar

`newMVar :: a -> IO (MVar a)` erzeugte initialisierte MVar

`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

`isEmptyMVar :: MVar a -> IO Bool` testet ob MVar leer ist

`tryTakeMVar :: MVar a -> IO (Maybe a)` non-blocking

`tryPutMVar :: MVar a -> a -> IO Bool` non-blocking

Beispiel für MVars: Rendevous

```
threadA :: MVar Int -> MVar Double -> IO ()
threadA valueToSendMVar valueReceiveMVar = do -- work
    putMVar valueToSendMVar 46 -- perform rendevous
    v <- takeMVar valueReceiveMVar
    putStrLn (show v )

threadB :: MVar Int -> MVar Double -> IO ()
threadB valueToReceiveMVar valueToSendMVar = do -- work
    -- perform rendevous by waiting on value
    z <- takeMVar valueToReceiveMVar
    putMVar valueToSendMVar (1.5 * (fromIntegral z))

main = do
  aMVar <- newEmptyMVar
  bMVar <- newEmptyMVar
  forkIO (threadA aMVar bMVar )
```

Explizite Synchronisation: MVars

MVars erlauben direkt

- gemeinsame Verwendung von Datenstrukturen, z.B. write-locks für Dateien
- 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
```

Software Transactional Memory

Software Transactional Memory (`STM`) erlaubt die Ausführung von IO-Threads auf gemeinsamen Variablen (`TVar`) ohne diese bei Beginn der Benutzung zu sperren ("lock-free").

Am Ende der Benutzung wird getestet ob Konflikte auftraten.

In dem Fall wird die Berechnung "zurückgespult".

Dazu müssen alle Operationen auf `TVar` in der `STM` Monade ausgeführt werden.

Zusammenfassung Explizite Parallelität

Modul **Control.Concurrent** erlaubt:

- Echte explizite Nebenläufigkeit
- Implizite Synchronisation über gemeinsame Teilausdrücke
- Explizite Synchronisation über `MVar` oder `Chan`
- "lock-free" Code auf gemeinsamen Variablen (`TVars`) in `STM` Monade
- Eignet sich zur Parallelisierung von monadischem Code

Par-Monade

Eine einfachere Alternative zu STM bietet die **Par-Monade** aus dem Modul **Control.Monad.Par**.

Nicht Teil der Haskell-Platform: `> cabal install monad-par`

- Berechnung bleibt voll deterministisch
  es kommt immer der gleiche Wert heraus
- Erlaubt nur Parallelität, aber keine Nebenläufigkeit
- Kümmert sich um globales Scheduling
- Deutlich teurer Overhead im Vergleich zu GpH-Sparks, d.h. für größere Einheiten einsetzen
- Kann nicht für Berechnungen mit IO-Operationen eingesetzt werden

Par-Monade

`Par` ist eine Instanz der Klasse `Monad`

```
runPar  :: Par a -> a
fork    :: Par () -> Par ()

new     :: Par (IVar a)
newFull :: NFData a => a -> Par (IVar a)
get     :: IVar a -> Par a
put     :: NFData a => IVar a -> a -> Par ()
```

- `runPar` führt die Monade aus
- `put` erzwingt die volle Auswertung seines Argumentes
- `fork` startet parallele Auswertung
- `get` wartet bis der Wert verfügbar ist
- `IVar` Variablen dürfen nur einmal beschrieben werden, Laufzeitfehler sonst

Beispiel

```
foo = runPar $ do
        x <- new
        y <- new
        fork $ put x left
        fork $ put y right
        resx <- get x
        resy <- get y
        return $ combine x y
where
  left         = ..Ausdruck mit aufwändiger Auswertung..
  right        = ..Ausdruck mit aufwändiger Auswertung..
  combine a b = ...
```

1. `IVar` Variablen anlegen
2. Parallele Berechnung starten
3. Auf Ende der Berechnung mit `get` warten

Beispiel

```
foo = runPar $ do
        x <- newFull left
        y <- newFull right


        resx <- get x
        resy <- get y
        return $ combine x y
  where
    left        = ..Ausdruck mit aufwändiger Auswertung..
    right       = ..Ausdruck mit aufwändiger Auswertung..
    combine a b = ...
```

`newFull` bietet bequeme Abkürzung zum Anlegen und Ausführen einer parallelen Berechnung

Zusammenfassung Par-Monade

- Ausschließlich zur Beschleunigung der Berechnung durch paralleles Auswerten
- Berechnung mit `Par`-Monade ist deterministisch, d.h. liefert immer das gleiche Resultat — nur evtl. schneller
- IO-Operationen innerhalb `Par`-Monade nicht erlaubt
- Erheblich mehr Verwaltungsaufwand als bei GpH, d.h. die parallel ausgeführten Berechnungseinheiten sollten alle wesentliche Berechnungen durchführen
  im Gegensatz zu Sparks wird jede parallele Einheit ausgeführt
- Parallele Einheiten werden immer vollständig ausgewertet
- Die `Par`-Monade kann jederzeit verwendet werden, ein Durchschleifen des Monaden-Typ ist nicht notwendig
  im Gegensatz zu `IO`