Sprachkonzepte der Parallelen Programmierung Vorlesung SS 11, WS 12

Johannes Waldmann, HTWK Leipzig

1. Februar 2013

Einleitung

Motivation

Herb Sutter: *The free lunch is over:* a fundamental turn towards concurrency in software. Dr. Dobb's Journal, März 2005.

CPUs werden nicht schneller, sondern bekommen mehr Kerne

2, 4 (i7-920), 6, 8, . . . 512 (GTX 580)

Wie programmiert man für solche Hardware?

Inhalt

- Primitiven und Abstraktionen zur Thread-Synchronisation: Semaphore, Monitore, Kanäle,
- thread-sichere Collections-Datentypen
- Transaktionen (Software Transactional Memory)
- deklarativer Parallelismus (Strategien)
- Rekursionsschemata für parallele Programme (skeletal parallelism)
- Anwendung: map/reduce-Framework
- impliziter Parallelismus: (Nested) Data Parallelism

Organisation

- jede Woche eine Vorlesung
- jede Woche eine Übung
  - gerade: Pool Z423, 2 Gruppen
  - ungerade: Seminar, beide Gruppen gemeinsam
- Prüfungsvorauss.: regelmäßiges und erfolgreiches Bearbeiten von Übungsaufgaben
- Prüfung: Klausur 120 min ohne Hilfsmittel

Literatur

- Maurice Herlihy und Nir Shavit: *The Art of Multiprocessor Programming*, Morgan Kaufmann 2008, `http://www.elsevierdirect.com/v2/companion.jsp?ISBN`
- Brian Goetz u.a.: *Java Concurrency in Practice*, Addison-Wesley 2006, `http://www.javaconcurrencyinpractice.com/`
- Brian O'Sullivan u.a.: *Real World Haskell*, O'Reilly 2008, `http://book.realworldhaskell.org/read/concurrent-an`
- Simon P. Jones: *Beautiful concurrency*, in: Wilson et al., *Beautiful Code*, O'Reilly 2007, `http://research.microsoft.com/en-us/um/people/simon`

Klassische Nebenläufigkeit

- Synchronisation von Prozessen (Threads) durch Sperren (Locks)
- dadurch Schutz kritischer Code-Abschnitte (für atomaren Zugriff auf gemeinsame Ressourcen)
- Realisierungen: z. B. wait/notify in Java
- die klassische Beispiel-Aufgabe: 5 Philosophen

Sicherer Datenaustausch

- gemeinsamer Speicherbereich, aber exklusive Zugriffe durch Locks
- Speicherzellen mit atomaren Zugriffen:

  Semaphore, MVar, AtomicLong
- lokale Parameterübergabe zwischen Co-Routinen

  Ada: Rendezvous (synchron)
- asynchroner Datentransport

  Aktoren, Nachrichten, Kanäle

Atomare Objekte, . . . in Datenstrukturen

- anstatt:

  ```
  long s; Object lock;
  synchronized (lock) { s += 1; }
  ```

  benutze `AtomicLong s; s.incrementAndGet();`
- um Zustandsänderung während *check-then-act* zu vermeiden:

  ```
  s.compareAndSet (expected, update);
  ```
- desgl. für `AtomicReference<V>`, benutzt in verketteten Datenstrukturen (Liste, Stack)
- Vorteil: keine globalen Locks, Nachteil: verteilter Zustand

Software Transactional Memory

Nachteil von Locks: Programmierung ist nicht modular.
Anderer Ansatz: spekulative Nebenläufigkeit:
Transaktionen mit optimistischer Ausführung

- innerhalb einer Transaktion: Protokollierung der Speicherzugriffe
- Abschluß (Commit) der Transaktion nur, wenn Protokoll konsistent ist
- sonst später erneut ausführen und Protokoll validieren
- eine abgebrochene Transaktion muß unbeobachtbar sein

Clojure: Transactions, Haskell: STM (das Typsystem hilft!)

Funktionales und paralleles Programmieren

(rein) funktionales Programmieren:

- keine (Neben-)Wirkungen, keine Zuweisungen,
- alle „Variablen" und „Objekte" sind konstant,
- nur Auswertung von Unterprogrammen,

ist trivial parallelisierbar und thread-sicher:
alle Argumente eines Unterprogrammes können parallel ausgewertet werden.

Parallele Auswertungsstrategien

Steuern der Auswertung durch Angabe von Strategien, unter Benutzung der Kombinatoren

- `par x y`: *Spark* für x, Resultat ist y
- `pseq x y`: auf x warten, dann Resultat y

Spark kann vom Laufzeitsystem gestartet werden (zu Thread konvertiert)

typische Anwendung: `par x ( pseq y ( f x y ))`

`http://hackage.haskell.org/packages/archive/`

Beispiel mergesort

Funktionales und paralleles Programmieren

Pro:

- leichte Parallelisierbarkeit für deklarativen Code
- deklaratives Modell für imperativen Code (MVar, STM)

Con:

- lazy evaluation
- garbage collection

aber:

- lazy evaluation ist selbst eine Form der Nebenläufigkeit (vgl. Iteratoren in OO)
- Nebenläufige garbage-collection wollen viele andere auch

Algorithmik

- welche Programme lassen sich gut (= flexibel) parallelisieren?
  (balancierter Berechnungsbaum, Tiefe anhängig von Prozessoren, nicht von Eingabe)
- welche Algorithmen kann man in dieser Form schreiben?
  (jedes fold über einen assoziativen Operator)
- wie findet man diese Operatoren, wie beweist man Assoziativität?

Beispiele:

- Summe der Zahlen einer Liste
- binäre Addition (Überträge!)
- Teilfolge mit maximaler Summe

Map/Reduce

Dean and Gemawat: *Simplified Data Processing on Large Clusters*, OSDI, 2004.
Ralf Lämmel: *Google's Map/Reduce Programming Model, Revisited*, in: Science of Computer Programming, 2006.
`http://userpages.uni-koblenz.de/~laemmel/Map`

```
mapReduce :: ( (k1,v1) -> [(k2,v2)] )
          -> ( k2 -> [v2] -> v3 )
          -> ( Map k1 v1 ) -> ( Map k2 v3 )
mapReduce m r
  = reducePerKey r -- 3. Apply r to each gro
  . groupByKey     -- 2. Group per key
  . mapPerKey m    -- 1. Apply m to each key
```


Threads, Thread-Sicherheit

Threads erzeugen und starten

Thread-Objekt implementiert `run()`, diese Methode wird aufgerufen durch `start()`.

```
for (int t=0; t<8; t++) {
    new Thread() {
        public void run() {
            System.out.println ("foo");
        }
    }.start();
}
```


Gemeinsamer Speicher, Synchronisation

dieses Beispiel zeigt die Probleme:

```
long s = 0; // gemeinsamer Speicher
for (int t=0; t<8; t++) {
    new Thread() {
        public void run() {
            for (...)  s += 1;
        } }.start(); }
// Synchronisation?
System.out.println (s);
```

Quelltext aus Vorlesung:
`git clone git://dfa.imn.htwk-leipzig.de/srv/`
browse:
`http://dfa.imn.htwk-leipzig.de/cgi-bin/gitwe`

Locks

jedes Java-Objekt kann als *lock* (Monitor, Sperre) dienen
`synchronized`-Blöcke: Betreten bei Lock-Besitz,
Verlassen mit Lock-Rückgabe,
für jeden Lock: zu jedem Zeitpunkt kann ihn höchstens ein Thread besitzen

```
Object lock = ...
synchronized (lock) { ... } // Anweisung

synchronized void m () { ... } // Methode
==> void m () { synchronized (this) { ... } }
```

Locks sind *re-entrant*, damit aus einer synchronisierten Methode eine andere aufgerufen werden kann (mit dem Lock, den der Thread schon besitzt)

Atomare Aktionen

- Operationen A_1 und A_2 sind *atomar zueinander*,

  wenn zu keinem Zeitpunkt ein Thread T_1 die Operation A_1 ausführt und gleichzeitig ein Thread T_2 die Operation A_2 ausführt.

- Operation A ist *atomar*,

  wenn sie atomar zu jeder anderen Operation ist (einschließlich sich selbst).

Object Confinement

Sichtbarkeit von Objekten (Objektverweisen) einschränken:

- Thread confinement: nur in einem Thread sichtbar,

  Beispiel: GUI-Frameworks (mit einem GUI-Thread, den der Programmierer der Applikation nie sieht)

- Stack confinement: Variable lebt nur während eines Methodenaufrufs

  (im Laufzeitkeller im Frame dieses Aufrufs)

gefährlich sind immer ungewollt exportierte Verweise, z. B. auf `this` im Konstuktor.

Einleitung, Definitionen

Eine Klasse heißt *thread-sicher*,

- wenn sie korrekt ist (= ihre Spezifikation erfüllt)
- auch bei Benutzung (Methodenaufruf) durch mehrere Threads mit beliebiger (durch das Laufzeitsystem ermöglichter) Ausführungsreihenfolge
- und ohne zusätzliche Synchronisation der Aufrufer.

thread-sichere Klassen synchronisieren selbst (Clients synchronisieren gar nicht)
zustandslose Klassen (Objekte) sind thread-sicher
(Brian Goetz et al.: *Java Concurrency in Practice*, A-W, 2006; Kap. 2/3)

Zusammengesetzte Aktionen

check-then-act

```
Stack<Foo> l = ... ;
if (! l.empty()) { Foo x = l.pop (); ... }
```

read-modify-write

```
int x = ... ;   x = x + 1 ;
```

sind nicht atomar und damit nicht thread-sicher
Auswege:

- Datentypen mit atomaren Operationen (AtomicLong) (später)
- Locking (jetzt)

Übung: *this* escapes during construction

- `class C { final int foo; ...}`

  Attribut `foo` wird erst im Konstruktor initialisiert

- der Konstruktor exportiert aber vorher `this`, dann kann das nicht initialisierte `foo` in einem anderen Thread beobachtet werden

- benutze
  `class Receiver { void receive (C x) { ... }`

- versteckter Export von `this`: als statischer Vorgänger einer lokalen Klasse (z. B. ActionListener)

Zustandsänderungen

wenn mehrere Threads eine gemeinsame Variable ohne Synchronisation benutzen, ist das Programm nicht thread-sicher.
Auswege:

- die Variable nicht an verschiedene Threads exportieren
- die Variable als unveränderlich (final) deklarieren
- Zugriffe synchronisieren

Übung: einfache Thread-Operationen in Java

- direkte Thread-Erzeugung mit

  `new Thread() { void run () { .. }}.start()`

- gegenseitiger Ausschluß durch

  `synchronized (lock) { .. }`

- Synchronisation durch `CountDownLatch`

- Thread-Erzeugung und -Verwaltung durch

  `void ExecutorService.execute(Runnable)`

- Synchronisation und Datentransport durch

  `Future<V> ExecutorService.submit(Callable<V`

PLTL-Spezifikationen von Systemeigenschaften

- gegenseitiger Ausschluß (*mutual exclusion*):

  Variablen: $p_i :=$ Prozeß i besitzt eine Ressource

  Spezifikation: $\Box\neg(p_1 \wedge \ldots \wedge p_n)$

- Fairness (kein Verhungern, *no starvation*)

  Variablen: $A_i :=$ Prozeß i beantragt Ressource; P_i

  Spezifikation: $\Box(A_1 \Rightarrow \Diamond P_1) \wedge \ldots \wedge \Box(A_n \Rightarrow \Diamond P_n)$

PLTL: propositional linear time logic

Syntax:

- Variablen $p, q, \ldots$, logische Operatoren $\neg, \vee, \wedge, \Rightarrow, \ldots$
- temporale Operatoren: immer $\Box$, irgendwann $\Diamond$,...

Beispiele: $\Diamond(p \vee q)$, $\Box\Diamond p$, $\Diamond\Box p$

Semantik: Wert der Formel F in Struktur K zur Zeit s:

- für $v \in V$: $\operatorname{wert}(v, K, s) = b_K(s)(v)$
- $\operatorname{wert}(F_1 \wedge F_2, K, s) = \min\{\operatorname{wert}(F_1, K, s), \operatorname{wert}(F_2, K, s)\}$
- $\operatorname{wert}(\Box F_1, K, s) = \min\{\operatorname{wert}(F_1, K, s') \mid s' \in \mathbb{N}, s' \geq s\}$
- $\operatorname{wert}(\Diamond F_1, K, s) = \max\{\operatorname{wert}(F_1, K, s') \mid s' \in \mathbb{N}, s' \geq s\}$

Übung: $\Diamond\Box\phi \Rightarrow \Box\Diamond\phi$ ist allgemeingülitg (gilt in jeder Struktur), ... aber die Umkehrung nicht

Kripke-Strukturen, Omega-Wörter

allgemein: Kripke-Struktur zu Variablenmenge V ist

- Graph (S, T) mit $S =$ Menge der Systemzustände, $T \subseteq S \times S$ Menge der Zustandsübergänge
- Knotenbeschriftung $b : S \to (V \to \mathbb{B})$ d.h., $b(s)$ ist eine Belegung der Variablen V

hier speziell:

- $S = \mathbb{N}$ (Zeitpunkte $0, 1, \ldots$)
- $T = \{(s, s+1) \mid s \in \mathbb{N}\}$ (linear time)

Beispiel:

- $V = \{p, q\}$,
- $b(s) = \{(p, (s \geq 3)), (q, (2 \mid s))\}$

Spezifikation und Verifikation nebenläufiger Prozesse

Einleitung

wie überall,

- Trennung von Spezifikation und Implementierung
- jeweils ein mathematisches Modell
- Sätze über Eigenschaften, Beziehungen dieser Modelle
- Algorithmen zur Beantwortung der Frage: erfüllt die Implementierung die Spezifikation?

so auch hier:

- Spezifikation: PLTL (propositional linear time logic)
- Implementierung: Omega-Wörter, -Sprachen, -Automaten

PLTL: Algorithmen

Satz: die folgenden Fragen sind entscheidbar:

- Modell-Problem:
  - Eingaben: eine PLTL-Formel F über V, ein schließlich periodisches Wort $w \in \Sigma^\omega$ mit $\Sigma = \mathbb{B}^V$
  - Frage: gilt $1 = \operatorname{wert}(F, w, 0)$
- Erfüllbarkeits-Problem:
  - Eingabe: eine PLTL-Formel F
  - Frage: gibt es $w \in \Sigma^\omega$ mit $1 = \operatorname{wert}(F, w, 0)$

Beweis-Idee: die Mengen $\{w \in \Sigma^\omega \mid 1 = \operatorname{wert}(F, w, 0)\}$ lassen sich durch endliche Automaten beschreiben.
(J. R. Büchi 1962, A. Pnueli 1977)

Omega-Wörter und -Sprachen

- jede lineare Kripke-Struktur über V

  entspricht einem unendlichen Wort über $\Sigma = 2^V$

  Bsp: $(0,1)(0,0)(0,1)(1,0)(1,1)(1,0)(1,1)\ldots$

- ein unendliches Wort (Omega-Wort) über Σ

  ist eine Abbildung $\mathbb{N} \to \Sigma$

- Σ^ω bezeichnet die Menge aller Omega-Wörter über Σ
- Schreibweise für Omega-Wörter mit schließlich periodischer Struktur:

  $(0,1)(0,0)(0,1)\ ((1,0)(1,1))^\omega$

  vgl. unendl. Dezimalbrüche $3/22 = 0.1\overline{36}$

Literatur

- Mordechai Ben-Ari: *Principles of Concurrent and Distributed Programming*, Prentice Hall 1990
- Beatrice Berard et al.: *Systems and Software Verification*, Springer 2001

erfordert eigentlich eine eigene Vorlesung, vergleiche

- Bertrand Meyer: *Concepts of Concurrent Computation*, `http://se.inf.ethz.ch/courses/2012a_spring/`
- Sibylle Schwarz: Verifikations- und Spezifikationsmethoden (Abschnitt 3: Model Checking) `http://whz-cms-10.zw.fh-zwickau.de/sibsc/le`

Granularität der Locks

- jede Zustandsvariable sollte durch genau einen Lock bewacht werden (im Quelltext dokumentieren!)
- Synchronisation einzelner Variablenzugriffe ist oft zu wenig
- Synchronisation einer gesamten Methode ist oft zu teuer (verhindert mögliche Nebenläufigkeit)

Für jede Klassen-Invariante: alle Variablen, die in der Invariante benutzt werden, müssen durch einen gemeinsamen Lock geschützt werden.

Semaphore

(allgemeiner) Semaphor ist abstrakter Datentyp
mit Zustand $S \in \mathbb{N}$ und *atomaren* Operationen:

- `Wait(S)` : wenn $S > 0$ dann $S := S - 1$, sonst blockiere
- `Signal(S)` : wenn es Prozesse gibt, die auf S warten, dann wecke einen davon auf, sonst $S := S + 1$

binärer Semaphor: $S \in \{0, 1\}$ und ...

- `Signal(S)` : ... sonst $S := 1$

E. W. Dijsktra: *Cooperating Sequential Processes, 4. The General Semaphore*, TU Eindhoven 1965
`http://www.cs.utexas.edu/~EWD/transcriptions`

Gegenseitiger Ausschluß (grundsätzlich)

```
Semaphore s := 1;
Gemeinsame Ressource r;
Prozeß Nr i { non_critical_section;
              Wait (s);
              cricital_section; // benutze r
              Signal (s);    }
```

Eigenschaften:

- gegenseitiger Ausschluß
- *fairness* für 2 Prozesse
- für ≥ 3 Prozesse nur *progress*
- fairness für ≥ 3, wenn blockierte Prozesse in Queue (statt Menge) verwaltet werden

Gegenseitiger Ausschluß (in Java)

jedes Objekt kann als Semaphor verwendet werden, dem vorigen Programm entspricht:

```
Object lock = new Object ();
Gemeinsame Ressource r;
Prozeß Nr i {
   non_critical_section
   synchronized (lock) { ... }
}
```

Ist eine Methode `synchronized` deklariert, dann ist `this` der Semaphor.

Namen für Semaphore

- hier definiert: Wait / Signal
- original Dijkstra: P / V
- java.util.concurrent.Semaphore: acquire / release
- java.lang.Object: wait / notify

Implizite und explizite Semaphore in Java

- für gegenseitigen Ausschluß während eines Methodenaufrufes:

  verwende `synchronized` (Semaphor ist implizit)

- für längere Zeiträume:

  `wait, notify` (Semaphor ist explizit)

Benutzung von `obj.wait(), obj.notify()` nur innerhalb von `synchronized (obj) { ... }`

Beispiel: Philosophen in der Mensa

(Edsger Dijkstra, Tony Hoare, ca. 1965)

- Prozess = Philosoph
- gemeinsame Ressource = Gabel

gewünschte System-Eigenschaften:

- liveness (kein Verklemmen)

  die Folge der Aktionen ist unendlich

- fairness (kein Verhungern)

  falls ein Prozeß eine Ressource anfordert, bekommt er sie nach endlich vielen Aktionen tatsächlich

Modellierung des Ressourcenzugriffs

Modellierung des ausschließlichen Ressourcenzugriffs:

```
class Fork {
    private boolean taken = false;
    synchronized void take () {
        while (taken) { wait (); }
        taken = true;               }
    synchronized void drop () {
        taken = false; notify ();   } }
```

beachte:

- beide Methoden sind `synchronized`
- `wait()` innerhalb einer Schleife, die die Bedingung testet (nach Aufwachen)

5 Philosophen: Aufgaben/Übung

Programmstruktur:

```
class Fork { void take() ; void drop () }
Philosoph i : new Thread () { void run () {
    this.nachdenken();
    fork[i].take(); fork[i+1].take();
    this.essen();
    fork[i].drop(); fork[i+1].drop();
}} . start();
```

welche Eigenschaften hat dieses Programm,

- global: progress oder deadlock?
- lokal: fairness?

wie kann man das ggf. reparieren?

Software Transactional Memory

Motivation/Plan

für nebenläufige Programme, die gemeinsamen Speicher benutzen:

- bisher: Synchronisation durch Sperren (locks)

  wesentlicher Nachteil: nicht modular

- jetzt: nichtblockierende Synchronisation

Quelle: Simon Peyton Jones: *Beautiful Concurrency*, = Kapitel 24 in: Andy Oram und Greg Wilson (Hrsg.): *Beautiful Code*, O'Reilly, 2007.
`http://research.microsoft.com/en-us/um/peopl`

Beispiel: Kontoführung (I)

das ist das (bisher) naheliegende Modell:

```
class Account { int balance;
  synchronized void withdraw (int m)
    { balance -= m; }
  synchronized void deposit (int m)
    { withdraw (-m); }
```

welche Fehler können hier passieren:

```
void transfer
      (Account from, Account to, int m)
{
  from.withdraw (m);
  to.deposit (m);
}
```


Beispiel: Kontoführung (II)

ist das eine Lösung?

```
void transfer
       (Account from, Account to, int m)
{
  from.lock(); to.lock ();
  from.withdraw (m);
  to.deposit (m);
  from.unlock(); to.unlock();
}
```


Beispiel: Kontoführung (III)

wann funktioniert diese Lösung und wann nicht?

```
  if (from < to) { from.lock(); to.lock() }
  else           { to.lock(); from.lock() }
  ...
```


Locks are Bad

- taking too few locks
- taking too many locks
- taking the wrong locks
- taking locks in the wrong order
- error recovery
- lost wakeups, erroneous retries

locks do not support modular programming
John Ousterhout: *Why Threads are a Bad Idea (for most puroposes)* USENIX 1996,
`http://www.cc.gatech.edu/classes/AY2009/cs42`

Speicher-Transaktionen

Benutzung:

- Transaktions-Variablen
- Lese- und Schreibzugriffe nur innerhalb einer Transaktion
- Transaktion wird atomar und isoliert ausgeführt

Implementierung:

- während der Transaktion: Zugriffe in Log schreiben
- am Ende (commit): prüfen, ob Log konsistent mit derzeitigem Speicherzustand ist
- . . . , wenn nicht, dann Transaktion wiederholen

Nebenwirkungen in Haskell: IO a

Werte:

```
4 :: Int  ;   "foo" ++ "bar" :: String
```

Aktionen mit Resultat und Nebenwirkung:

```
writeFile "foo.text" "bar" :: IO ()
readFile "foo.text" :: IO String
putStrLn (show 4) :: IO ()
```

Nacheinanderausführung von Aktionen:

```
do s <- readFile "foo.text"
   putStrln (show (length s))
```

Start einer Aktion: im Hauptprogramm

```
main :: IO ()
main = do ...
```


Nebenwirkungen auf den Speicher

```
import Data.IORef
data IORef a -- abstrakt
newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

- damit kann man die üblichen imperativen Programme schreiben (jede Variable ist eine IORef)
- die Kombinatoren zur Programmablaufsteuerung kann man sich selbst bauen, z. B.

  ```
  while :: IO Bool -> IO () -> IO ()
  ```

  Übung: while implementieren, Fakultät ausrechnen

Transaktionen: STM a

jede Transaktion soll *atomar sein*
⇒ darf keine IO-Aktionen enthalten (da man deren Nebenwirkungen sofort beobachten kann)
neuer Typ `STM a` für Aktionen mit Nebenwirkungen *nur auf Transaktionsvariablen* `TVar a`

```
type Account = TVar Int
withdraw :: Account -> Int -> STM ()
withdraw account m = do
    balance <- readTVar account
    writeTVar account ( balance - m )
transfer :: Account -> Account -> Int -> IO
transfer from to m = atomically
  ( do withdraw from m ; deposit to m    )
```


Bedingungen und Auswahl

- eine Transaktion abbrechen: `retry`
- eine Transaktion nur ausführen, wenn eine Bedingung wahr ist

  ```
  check :: Bool -> STM ()
  check b = if b then return () else retry
  ```

- eine Transaktion nur ausführen, wenn eine andere erfolglos ist: `orElse`

STM-Typen und -Operationen

```
data STM a -- Transaktion mit Resultat a
data IO  a -- (beobachtbare) Aktion
           -- mit Resultat a
atomically :: STM a -> IO a
retry      :: STM a
orElse     :: STM a -> STM a -> STM a

data TVar a -- Transaktions-Variable
            -- mit Inhalt a
newTVar    :: a -> STM ( TVar a )
readTVar   ::
writeTVar  ::
```

(= Tab. 24-1 in Beautiful Concurrency)

vgl.
`http://hackage.haskell.org/packages/archive/`

The Santa Claus Problem

Santa repeatedly sleeps until wakened by either all of his nine reindeer, back from their holidays, or by a group of three of his ten elves. If awakened by the reindeer, he harnesses each of them to his sleigh, delivers toys with them and finally unharnesses them (allowing them to go off on holiday). If awakened by a group of elves, he shows each of the group into his study, consults with them on toy R&D and finally shows them each out (allowing them to go back to work). Santa should give priority to the reindeer in the case that there is both a group of elves and a group of reindeer waiting.
J.A. Trono, *A new Exercise in Concurrency*, SIGCSE Bulletin 26, 1994, p. 8-10

Philosophen mit STM

kein Deadlock (trivial).

```
forM [ 1 .. num ] $ \ p -> forkIO $ forever
    atomically $ do
        take $ left  p
        take $ right p
    atomically $ drop $ left  p
    atomically $ drop $ right p
take f = do
    busy <- readTVar f
    when busy $ retry
    writeTVar f True
```

nicht fair. Vergleiche Diskussion hier:
`http://thread.gmane.org/gmane.comp.lang.hask`

Nicht blockierende Synchronsiation

Einleitung

Synchronisation (geordneter Zugriff auf gemeinsame Ressourcen) durch

- explizite Sperren (lock)
  pessimistische Ausführung
  Gefahr von Deadlock, Livelock, Prioritätsumkehr
- ohne Sperren (lock-free)
  optimistische Ausführung
  ein Prozeß ist erfolgreich (andere müssen wiederholen)
  – nur feingranular (`AtomicLong`, `compareAndSet()`)
  – atomare zusammengesetze Transaktionen

Literatur

- *Atomic Variables and Nonblocking Synchronization*, Kapitel 15 in Brian Goetz et al.: *Java Concurrency in Practice*
- *Counting, Sorting and Distributed Coordination*, Kapitel 12 in Maurice Herlihy and Nir Shavit: *The Art of Multiprocessor Programming*
- Which CPU architectures support Compare And Swap (CAS)?

  `http://stackoverflow.com/questions/151783/`

Compare-and-Set (Benutzung)

Der Inhalt einer Variablen soll um 1 erhöht werden.
Mit STM wäre es leicht:

```
atomically $ do
    v <- readTVar p ; writeTVar p $! (v+1)
```

ohne STM, mit einfachen atomaren Transaktionen:

```
AtomicInteger p;  boolean ok;
do { int v = p.get();
     ok = p.compareAndSet(v,v+1);
} while ( ! ok);
```

- Vorteil: das geht schnell (evtl. sogar in Hardware)
- Nachteil: nicht modular (keine längeren Transaktionen)
- Auswirkung: kompliziertere Algorithmen

Compare-and-Set (Implementierung)

Modell der Implementierung:

```
class AtomicInteger {  private int value;
  synchronized int get () { return value; }
  synchronized boolean
    compareAndSet (int expected, int update)
      if (value == expected) {
        value = update ; return true;
      } else {
        return false; } } }
```

moderne CPUs haben CAS (oder Äquivalent)
im Befehlssatz (Ü: suche Beispiele in x86-Assembler)

JVM (ab 5.0) hat CAS für Atomic{Integer,Long,Reference}

Compare-and-Set (JVM)

Assembler-Ausgabe (des JIT-Compilers der JVM):

```
javac CAS.java
java -Xcomp -XX:+UnlockDiagnosticVMOptions -
```

`http://wikis.sun.com/display/HotSpotInternal`

Vorsicht, Ausgabe ist groß. Mit `nohup` in Datei umleiten, nach `AtomicInteger.compareAndSet` suchen.

auch nützlich:
`http://blogs.sun.com/watt/resource/jvm-optio`

Non-Blocking Stack

Anwendung: Scheduling-Algorithmen:
(jeder Thread hat Stack mit Aufgaben, andere Threads können dort Aufgaben hinzufügen und entfernen)

```
private static class Node<E> {
  E item; Node<E> next;
}
class Stack<E> {
  AtomicReference<Node<E>> top
    = new AtomicReference<Stack.Node<E>>();
  public void push (E x)
  public E pop ()
}
```


Non-Blocking Queue (Problem)

- einfach verkettete Liste

```
private static class Node<E> {
   E item; AtomicReference<Node<E>> next; }
```

- Zeiger `head,tail` auf Anfang/Ende, benutze Sentinel (leerer Startknoten)

Auslesen (am Anfang) ist leicht,
Problem beim Einfügen (am Ende):

- zwei Zeiger `next` und `tail` müssen geändert werden,
- aber wir wollen keinen Lock benutzen.

Non-Blocking Queue (Lösung)

(Michael and Scott, 1996)
`http://www.cs.rochester.edu/research/synchro`

Idee: die zwei zusammengehörigen Änderungen mglw. durch verschiedene Threads ausführen (!)

Queue hat zwei Zustände:

- A: tail zeigt auf letzten Knoten
- B: tail zeigt auf vorletzten Knoten

wer B bemerkt, muß reparieren.

in Java realisiert als ConcurrentLinkedQueue

Verteiltes Zählen

Motivation: viele Algorithmen benötigen eine zentrale Ausgabe von Tickets (mit eindeutigen und aufsteigenden Nummern).

Das wäre ein Flaschenhals (bei vielen Prozessen):

```
class Counter { int count;
synchronized int next () { return count++;}}
```

interessante und überraschende Lösung: *Zählnetzwerke*
James Aspnes, Maurice Herlihy, and Nir Shavit.
Counting networks, JACM 41(5):1020–1048, Sept. 1994
`http://www.cs.yale.edu/homes/aspnes/papers/a`

wesentlicher Baustein: `AtomicBoolean.negate()`

Verteiler, Netzwerke

Def: ein *Verteiler* (balancer) ist Schaltkreis mit zwei Eingängen, zwei Ausgängen, einem Zustand.
Wenn Zustand *hoch*, erscheint nächstes Eingangstoken am oberen Ausgang. Wenn Zustand *tief*, am unteren.
Nach jedem Token wechselt der Zustand.

Def: ein n-*Netzwerk* hat n Eingänge und n Ausgänge und besteht aus Verteilern.

Def: ein n-Netzwerk ist ein n-*Zählnetzwerk* (ZN),
wenn für *jede* Eingabe von Token an Eingängen
und für *jede* nebenläufige Verarbeitung im Netzwerk
die Token an den Ausgängen $1, 2, \ldots, n, 1, 2, \ldots$ erscheinen.
jeder Verteiler ist ein 2-ZN. — gibt es 4-ZN? größere?
Wie baut man daraus einen verteilten Zähler?

Bitonisches Zählen und Zusammenfügen (I)

Def: eine Zahlenfolge $[x_1, \ldots, x_n]$ heißt *Schrittfolge*, wenn $x_1 \geq \ldots \geq x_n \geq x_1 - 1$.
Anwendung: Für y_i= Anzahl der Token an Ausgang i gilt: N ist Zählnetz $\iff$ für jede Eingabe ist (y_i) eine Schrittfolge.

Ansatz: definiere Teilnetzwerke M, deren Eingangsfolgen (nach Induktion) Schrittfolgen sind.

Konstruktion der Zählnetze: Induktionsanfang: $C_1(x_1) =$
Induktionsschritt: $C_{2n}(x_1, \ldots, x_{2n}) =$
$C_n(x_1, \ldots, x_n); C_n(x_{n+1}, \ldots x_{2n}); M_{2n}(x_1, \ldots, x_n; x_{n+1}, \ldots, x_{2n})$

Konstruktion der Merge-Netze: (Spezifikation?)
Induktionsanfang: $M_2(x_1, x_2)$; Induktionsschritt?

Bitonisches Zählen und Zusammenfügen (II)

Induktionsschritt:

$$M_{2n}(\vec{x}, \vec{y}) = \begin{cases} M_n(\operatorname{odd} \vec{x}, \operatorname{even} \vec{y}); \\ M_n(\operatorname{even} \vec{x}, \operatorname{odd} \vec{y}); \\ V(x_1, x_2); \ldots; V(y_{n-1}, y_n) \end{cases}$$

mit $V(p, q)$ = Verteiler, $\operatorname{odd}(x_1, x_2, \ldots) = (x_1, x_3, \ldots)$, $\operatorname{even}(x_1, x_2, \ldots) = (x_2, x_4, \ldots)$.

Satz: jedes solche M_n erfüllt die Spezifikation.

Übung: konstruiere C_4, M_4
Übung: Beweis für M_8 mit Eingangsfolge $(3, 3, 2, 2; 4, 3, 3, 3)$, unter der Annahme, daß der Satz für M_4 gilt.
Übung: Beweis für M_{2n} mit beliebiger Eingangsfolge, unter der Annahme, daß der Satz für M_n gilt.

Bitonisches Zählen und Zusammenfügen (Aufgaben)

Beweise: die folgenden Bedingungen sind äquivalent:

- $(x_1, \ldots, x_n)$ ist Schrittfolge
- $\forall 1 \leq i < j \leq n : 1 \geq x_i - x_j \geq 0.$
- Wenn $m = \sum x_i$, dann $\forall i : x_i = \lceil \frac{m-i+1}{n} \rceil$

Wenn x eine Schrittfolge ist, welche Beziehungen gelten zwischen $\sum \operatorname{odd}(x), \sum(x)/2, \sum \operatorname{even}(x)$?
(Möglichst genau! Benutze ggf. $\lceil\cdot\rceil, \lfloor\cdot\rfloor$

Beweise: Wenn x und y gleichlange Schrittfolgen mit $\sum x = 1 + \sum y$, dann gilt für alle bis auf ein i: $x_i = y_i$.
Was gilt stattdessen für dieses i?

Lokale Prozeßkommunikation

Motivation

bisher betrachtete Modelle zur Thread-Kommunikation:

- Datenaustausch über gemeinsamen Speicher
- Synchronisation durch Locks, Transaktionen

jetzt:

- kein gemeinsamer Speicher
- Datentransport durch Nachrichten
- dabei ggf. Synchronisation

Beispiel: Rendezvous (Ada), Actors (Scala), Channels (Go)

Communicating Sequential Processes (CSP)

- abstraktes Modell für Kommunikation von Prozessen
- Abstraktion: (endliches) Alphabet von (einfachen) Nachrichten, synchrone Kommunikation
- entwickelt 1978 von C. A. R. Hoare

  `http://research.microsoft.com/en-us/people/`
- Grundlage für Prozeßmodell in Occam, Ada, Go, ...

CSP: Syntax

E ist eine Menge von Ereignissen
Die Menge $\mathbb{P}(E)$ der Prozesse über E definiert durch:

- STOP $\in \mathbb{P}$,
- wenn $e \in E$ und $P \in \mathbb{P}$, dann $(e \to P) \in \mathbb{P}$
- wenn $P_1, P_2 \in \mathbb{P}$, dann sind in $\mathbb{P}$:
  - $P_1; P_2$ (Nacheinanderausführung)
  - $P_1 \Box P_2$ (Auswahl)
  - für $C \subseteq E$: $P_1 \|_C P_2$ (nebenläufige Ausführung mit Kommunikation)
- $P_1^* \in \mathbb{P}$ (eine Form der Iteration)

CSP: Semantik (Spur-Semantik)

zu $P \in \mathbb{P}(E)$ konstruiere Automaten A (mit ϵ-Übergängen) über E.
Die *Spur-Sprache* von $P :=$ die Sprache von A
Startzustand von A ist P, Übergangsrelation von A ist:

- $(a \to P) \xrightarrow{a} P$
- $(\text{STOP}; Q) \xrightarrow{\epsilon} Q$, wenn $P \xrightarrow{a} P'$, dann $(P; Q) \xrightarrow{a} (P'; Q)$,
- $(P \Box Q) \xrightarrow{\epsilon} P$, $(P \Box Q) \xrightarrow{\epsilon} Q$
- $a \in C \wedge P \xrightarrow{a} P' \wedge Q \xrightarrow{a} Q' \Rightarrow (P \|_C Q) \xrightarrow{a} (P' \|_C Q')$,
- $a \notin C \wedge P \xrightarrow{a} P' \Rightarrow (P \|_C Q) \xrightarrow{a} (P' \|_C Q)$,
- $a \notin C \wedge Q \xrightarrow{a} Q' \Rightarrow (P \|_C Q) \xrightarrow{a} (P \|_C Q')$,
- $P^* \xrightarrow{\epsilon} \text{STOP}$, $P^* \xrightarrow{\epsilon} (P; P^*)$.

Rendez-Vous (I) in Ada

```
task body Server is
   Sum : Integer := 0;
begin loop
      accept Foo (Item : in Integer)
        do Sum := Sum + Item; end Foo;
      accept Bar (Item : out Integer)
        do Item := Sum; end Bar;
   end loop;
end Server;
A : Server; B : Integer;
begin
   A.Foo (4); A.Bar (B); A.Foo (5); A.Bar (B
end B;
```

Rendezvous (II)

- ein Prozeß (Server) führt `accept` aus,
  anderer Prozeß (Client) führt Aufruf aus.
- beide Partner müssen aufeinander warten
- `accept Foo ( .. ) do .. end Foo` ist atomar

Rendezvous (III)

allgemeinere Formen von `accept`:

```
• select accept Foo (Item : in Integer)
            do .. end;
     or accept Bar ( ... )
  end select;
```

```
• when X < Y => accept Foo (...)
  select ... or terminate; end select;
  select ... or delay 1.0 ; ... end select;
  select ... else .. end select;
```

http://en.wikibooks.org/wiki/Ada_Programming
http://www.adaic.org/resources/add_content/s

Kommunikations-Kanäle

zur asynchronen Kommunikation
(Eigenschaften: vgl. Postbrief statt Rendezvous)

Designfragen:

- Kapazität des Kanals/Briefkastens
  (Kapazität 0 $\Rightarrow$ Rendezvous)
- Ordnung der Nachrichten (FIFO oder ungeordnet)

Bsp. in Go: (`http://golang.org`)

```
ch := make (chan int) // anlegen
ch <- 41 // schreiben
x := <- ch // lesen
```


Actors (Scala)

http://www.scala-lang.org/node/242

```
object Stop
class Server extends Actor { def act() {
    var running = true;
    while (running) { receive {
        case x : Int => println(x)
        case Stop => running = false; } } }
var s = new Server()
s.start ; s ! 42 ; s ! Stop
```


Good Actors Style

Kap. 30.5 in: Odersky, Spoon, Villers: *Programming in Scala*, Artima 2007,

- ein Aktor soll nicht blockieren
  ...sondern lieber Arbeit an andere Aktoren weitergeben
- kommuniziere mit Aktoren nur durch Nachrichten
  ...und nicht durch gemeinsame Variablen
- Nachrichten sollten *immutable* sein
  ...sonst Gefahr von inkonsistenten Daten
- Nachrichten sollten *self-contained* sein
  ...damit der Aktor nicht nachfragen muß
  unveränderliche Objekte kann man billig mitschicken

Rendezvous-Zusammenfassung

- unmittelbar synchron, kein Puffer:
  - Ada-Rendezvous (task entry call/accept)
  - Go:
    `ch = make(chan int); ch <- .. ; .. <- ch`
  - Scala: `Actor a ; ... = a !? msg`
- gepuffert synchron (Nachrichten der Reihe nach)
  - beschränkte Kapazität:
    Go: `make(chan int, 10)`
    `java.util.concurrent.LinkedBlockingQueue`
  - unbeschränkt:
    Haskell: `Control.Concurrent.newChan`
- asynchron Scala: `Actor a ; ... = a ! msg`

Verteilte Programme

Verteiltes Rechnen

- Prozesse mit gemeinsamem Speicher
- Prozesse (Aktoren), Nachrichten/Kanäle
- Prozesse (Aktoren) verteilt auf verschiedene Rechner

Realisierungen:

- Erlang (1987...)
- Cloud Haskell (2012...)

Erlang

*Er*icsson *Lang*uage, `http://www.erlang.org/`
Anwendung: Steuerung von Telekommunikationsanlagen

grundsätzliche Spracheigenschaften:

- funktional
- dynamisch typisiert
- mit Nebenwirkungen

(also ungefähr LISP)

Besonderheiten:

- leichtgewichtige verteilte Prozesse
- *hot code replacement* (paßt gut zu *tail recursion*)

Cloud Haskell: Übersicht

http://www.haskell.org/haskellwiki/Cloud_Haskell

- keine Sprache, sondern Bibliothek

  (= eDSL, *eingebettete* domainspezifische Sprache)

- Semantik angelehnt an Erlang-Prozesse

Jeff Epstein, Andrew Black, and and Simon Peyton Jones. *Towards Haskell in the Cloud*, Haskell Symposium, Tokyo, Sept 2011.
http://research.microsoft.com/en-us/um/people/simonpj

Cloud-Haskell: elementare Operationen

```
findSlaves :: Backend -> Process [NodeId]
spawn :: NodeId -> Closure (Process ())
      -> Process ProcessId
send :: Serializable a
     => ProcessId -> a -> Process ()
expect :: Serializable a => Process a

newChan :: Serializable a
    => Process (SendPort a, ReceivePort a)
sendChan :: Serializable a
    => SendPort a -> a -> Process ()
receiveChan :: Serializable a
    => ReceivePort a -> Process a
```


Übungsaufgabe zu Prozessen

50 Gefangene

(Quelle: Bulletin EATCS, Puzzle Corner, 200?)

- Ein Zimmer mit Lampe + Schalter.
- Der *faire* Wärter führt Gefangene einzeln in das Zimmer.
- Wenn jemand STOP ruft, und zu dem Zeitpunkt alle ≥ 1 mal im Zimmer waren, dann werden alle freigelassen.

Spezifikation: I_k : ist-drin, W_k : war-drin
$(\neg W_k \mathsf{U} I_k) \wedge \Box(I_k \Rightarrow \Box W_k)$
Fairness: $\bigwedge_k : \Box\Diamond I_k$.   impliziert Ziel: $\Diamond \bigwedge_k W_k$
Aufgabe: 1. formale Semantik von U (until)?
2. Lösung? (2a Lampe ist anfangs aus, 2b … unbekannt)

Denksport

- 3 Mann mit Hut
  2 blaue und 3 rote Hüte, jeder sieht jeden anderen,
  Aussagen: „weiß nicht, weiß nicht, weiß"
  Verallg. für mehr Personen? mehr Farben?
- haben alle Stücke die gleiche Masse?
  a) 8 Stücke, 2 Massen, 3 Wägungen. b) 10 Stücke
  Quelle: *Puzzled* in: Comm. ACM 11/2012
- 45 min abmessen
  mit zwei inhomogenen Lunten, die 60 min brennen
- Bergsteiger
  200 m tief, 150 m Seil und Messer, Haken bei 100 m
- 2 Uhren, 100 Stufen

Parallele Auswertungsstrategien

Überblick

- bei Ausdrücken $f(X, Y)$ kann man Werte von X und Y parallel und unabhängig berechnen,
- wenn die Auswertung von X und Y nebenwirkungsfrei ist.
- im einfachsten Fall sind *alle* Ausdrücke nebenwirkungsfrei (Haskell)
- parallele Auswertung durch
  - Kombinatoren: `par X (pseq Y (f X Y))`
  - Strategie-Annotationen: `` xs `using` parList rseq ``
- Haskell benutzt Bedarfsauswertung, diese muß man ggf. umgehen, d.h. Auswertung von Teilausdrücken erzwingen

Algebraische Datentypen und Pattern Matching

ein Datentyp mit zwei Konstruktoren:

```
data List a
   = Nil              -- nullstellig
   | Cons a (List a)  -- zweistellig
```

Programm mit Pattern Matching:

```
length :: List a -> Int
length xs = case xs of
    Nil       -> 0
    Cons x ys -> 1 + length ys
```

beachte: Datentyp rekursiv ⇒ Programm rekursiv

```
append :: List a -> List a -> List a
```


Alg. Datentypen (Beispiele)

```
data Bool = False | True
data Maybe a = Nothing | Just a
data Tree a =
    Leaf | Branch ( Tree a ) a ( Tree a)
```

Ü: inorder, preorder, leaves, depth
Ü: Schlüssel in Blättern

```
data N = Z | S N
```

Ü: Rechenoperationen

Notation für Listen in Haskell:
anstatt `data List a = Nil | Cons a (List a)`
wird benutzt `data [a] = [] | (a : [a])`

Bedarfsauswertung

- Konstruktoren werten Argumente (zunächst) nicht aus

  statt Wert wird *thunk* (closure) gespeichert

- Wert eines Ausdrucks wird erst bestimmt, wenn er wegen Pattern Matching benötigt wird

- dann wird der Wert nur soweit nötig bestimmt

  d. h., bis man den obersten Konstruktor sieht

  eine solche Form heißt *Kopfnormalform*

(der andere Begriff ist *Normalform*: alle Konstruktoren)

Beispiel: Primzahlen

Aufgabe: bestimme $\pi(n) :=$ Anzahl der Primzahlen in $[1..n]$ auf naive Weise (durch Testen und Abzählen)

```
num_primes_from_to :: Int -> Int -> Int
num_primes_from_to lo hi
    = length $ filter id
    $ map prime [ lo .. hi ]
prime :: Int -> Bool
```

parallele Auswertung von `map prime [lo..hi]`:

```
map prime [lo..hi]
    `using` parListChunk 100000 rseq
```

`http://hackage.haskell.org/packages/archive/parallel/`

Beispiel: Mergesort

Sequentieller Algorithmus:

```
merge :: Ord a => [a] -> [a] -> [a]
split :: [a] -> ([a],[a])
msort :: Ord a => [a] -> [a]
msort [] = [] ; msort [x] = [x] ; msort xs =
    let ( here, there ) = split xs
        mshere  = msort here
        msthere = msort there
    in  merge mshere msthere
```

Strategie-Annotation in `msort`,
dabei Auswertung der Teilresultate erzwingen.
vgl.
`http://article.gmane.org/gmane.comp.lang.has`

Parallel LINQ

Beispiel:

```
(from n in Enumerable.Range(lo, hi-lo)
                   .AsParallel()
 where Prime(n)  select true).Count ();
```

Typen:

- `System.IEnumerable<E>`
- `System.Linq.ParallelEnumerable<E>`

`http://msdn.microsoft.com/en-us/library/dd99`

Übung:

- paralleles `foreach`
- Steuerung der Parallelität durch `Partitioner`

Verifikation funktionaler Programme

Term-Gleichungen

funktionales Programm = Gleichungssystem

- Grundbereich ist Menge der Terme (Bäume)
- Gleichungen sind die Funktionsdefinitionen

Beweis von Programm-Eigenschaften durch

- *equational reasoning*

  (äquivalentes Umformen, Ersetzen von Teiltermen durch gleichwertige)
- Induktion (nach dem Termaufbau)

Append ist assoziativ

```
data List a = Nil | Cons a ( List a )
    deriving (Show, Eq)
append :: List a -> List a -> List a
append xs ys = case xs of
    Nil        -> ys
    Cons x xs' -> Cons x ( append xs' ys )
```

Behauptung:

```
forall a :: Type, forall xs, ys, zs :: List
  append xs (append ys zs)
    == append (append xs ys) zs
```

Beweis:
Fall 1: `xs = Nil` (Induktionsanfang) Fall 2:
`xs = Cons x xs'` (Induktionsschritt)

Fall 1: `xs = Nil` (Induktionsanfang)

```
append Nil (append ys zs)
   =?= append (append Nil ys) zs
(append ys zs) =?= append (append Nil ys) zs
(append ys zs) =?= append      ys zs
```

Terme sind identisch
Fall 2: `xs = Cons x xs'` (Induktionsschritt)

```
append (Cons x xs') (append ys zs)
   =?= append (append (Cons x xs') ys) zs
Cons x (append xs' (append ys zs))
   =?= append (Cons x (append xs' ys)) zs
Cons x (append xs' (append ys zs))
   =?= Cons x (append (append xs' ys) zs)
```

Teilterme sind äquivalent nach Induktionsannahme

Verifikation — Beispiele

- `append :: List a -> List a -> List a`
  ist assoziativ
- für `reverse :: List a -> List a` gilt:
  `reverse (reverse xs) == xs`
- Addition von Peano-Zahlen ist kommutativ

Rekursionsmuster

Prinzip:

- rekursive Datenstruktur (algebraischer Datentyp)

⇒ Rekursionsmuster für Algorithmen, die diese Struktur benutzen (verarbeiten).

Implementierungen:

- map/fold in Haskell (funktional)
- Besucher für Kompositum (objektorientiert)
- Select/Aggregate in C# (funktional)

Anwendung in paralleler Programmierung:

- gewisse Muster lassen sich flexibel parallelisieren

Rekursion über Bäume (Beispiele)

```
data Tree a    = Leaf
  | Branch ( Tree a ) a ( Tree a )
summe :: Tree N -> N
summe t = case t of
  Leaf -> Z
  Branch l k r ->
    plus (summe l) (plus k (summe r ))
preorder :: Tree a -> List a
preorder t = case t of
  Leaf -> Nil
  Branch l k r ->
   Cons k (append (preorder l)
                  (preorder r))
```


Rekursion über Bäume (Schema)

gemeinsame Form dieser Funktionen:

```
f :: Tree a -> b
f t = case t of
  Leaf         -> ...
  Branch l k r -> ... (f l) k (f r)
```

dieses Schema ist eine Funktion höherer Ordnung:

```
fold :: ( ... ) -> ( ... ) -> ( Tree a -> b
fold leaf branch = \ t -> case t of
  Leaf -> leaf
  Branch l k r ->
     branch (fold leaf branch l) k
            (fold leaf branch r)
summe = fold Z ( \ l k r -> plus l (plus k r
```


Rekursion über Listen

```
and :: List Bool  -> Bool
and xs = case xs of
     Nil -> True ; Cons x  xs' -> x && and x
length :: List a -> Int
length xs = case xs of
    Nil -> 0 ; Cons x xs' ->  1 + length xs'

fold :: b -> ( a -> b -> b ) -> [a] -> b
fold nil cons xs = case xs of
    Nil -> nil
    Cons x xs' -> cons x ( fold nil cons xs'
and = fold True (&&)
length = fold 0 ( \ x y ->  1 + y)
```


Rekursionsmuster (Prinzip)

jeden Konstruktor durch eine passende Funktion ersetzen.

```
data List a = Nil | Cons a (List a)
fold ( nil :: b ) ( cons :: a -> b -> b )
   :: List a -> b
```

Rekursionsmuster instantiieren = (Konstruktor-)Symbole interpretieren (durch Funktionen) = eine Algebra angeben.

```
length  = fold 0   ( \ _ l -> l + 1 )
reverse = fold Nil ( \ x ys ->      )
```


Rekursion über Listen (Übung)

Aufgaben:

- `append, reverse, concat, inits, tails` mit `fold` (d. h., ohne Rekursion)

Bezeichnungen in Haskell-Bibliotheken:
das vordefinierte Rekursionsschema über Listen ist:

```
foldr :: (a -> b -> b) -> b -> ([a] -> b)
length = foldr ( \ x y -> 1 + y ) 0
```

Beachte:

- Argument-Reihenfolge (erst cons, dann nil)
- foldr nicht mit foldl verwechseln (foldr ist das „richtige“)

Fold/Besucher in C#

fold für Listen = System.Linq.Aggregate

```
import System.Linq;
import System.Collections.Generic;

List<int> l =
    new List<int>() { 3,1,4,1,5,9 };
Console.WriteLine
    (l.Aggregate(0, (x,y) => x+y));
```


Homomorphiesätze

Begriffe (allgemein)

homo-morph = gleich-förmig
Signatur Σ (= Menge von Funktionssymbolen)
Abbildung h von Σ-Struktur A nach Σ-Struktur B ist Homomorphie, wenn:
$\forall f \in \Sigma, x_1, \ldots, x_k \in A$:
$$h(f_A(x_1, \ldots, x_k)) = f_B(h(x_1), \ldots, h(x_k))$$
Beispiel:
Σ = Monoid (Eins-Element 1, binäre Operation $\circ$)
$A = \mathrm{List\ a}$ (Listen) mit $1_A = \mathrm{Nil}, \circ_A = \mathrm{append}$
$B = \mathrm{N}$ (Zahlen) mit $1_B = \mathrm{Z}, \circ_A = \mathrm{plus}$
$h = \mathrm{length}$

Homomorphie-Sätze für Listen

1. für jeden Hom exist. Zerlegung in map und reduce — und das reduce kann man flexibel parallelisieren!

   Bsp: `length = reduce (+) . map (const 1)`

   `map`: parallel ausrechnen, `reduce`: balancierter Binärbaum.
2. jeden Hom. kann man als foldl und als foldr schreiben
3. (Umkehrung von 2.) Wenn eine Funktion sowohl als foldl als auch als foldr darstellbar ist, dann ist sie ein Hom. — und kann (nach 1.) flexibel parallelisiert werden

   m.a.W: *aus der Existenz zweier sequentieller Algorithmen folgt die Existenz eines parallelen Alg.*

Literatur

- Jeremy Gibbons: *The Third Homomorphism Theorem*, Journal of Functional Programming, May 1995.

  `http://citeseerx.ist.psu.edu/viewdoc/downlc`
- Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, Masato Takeichi: *Automatic Inversion Generates Divide-and-Conquer Parallel Programs*, PLDI 2007.

foldr, foldl, map, reduce

- Rechnung beginnt am rechten Ende

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr (-) 0 [1,2,3] = 2
```

- Rechnung beginnt am linken Ende:

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl (-) 0 [1,2,3] = -6
```

- für assoziative Operation, beliebige Klammerung:

```
reduce :: b -> (b -> b -> b) -> [b] -> b
```

- mit elementweiser Transformation (map):

```
foldb :: b->(a->b)->(b->b->b)-> [a] -> b
map   :: (a -> b) -> [a] -> [b]
foldb n f g xs = reduce n g ( map f xs )
```


Beispiel: maximale Präfix-Summe

```
mps :: [Int] -> Int
mps xs = maximum
    $ do ys <- inits xs ; return $ sum ys
```

zur Darstellung durch fold(l/r): benutze

```
mpss :: [ Int ] -> ( Int, Int )
mpss xs = ( mps xs, sum xs )
```

Bestimme

- `mpss (x : xs)` aus `mpss xs` ergibt `mpss = foldr ...`
- `mpss (xs ++ [x])` aus `mpss xs` ergibt `mpss = foldl ...`

nach 3. Homomorphiesatz existiert `mpss = foldb ...`

Schwache Inverse

- Def: f' heißt *schwach invers* zu f, wenn $\forall x : f(f'(f(x))) = f(x)$.

  Bsp: `sum' :: ... -> ... ; sum' x = ...`
- Ziel: `f = foldb ... h` mit

  `h x y = f (f' x ++ f' y)`
- Satz: diese Darstellung existiert und ist korrekt, wenn f sowohl als foldr als auch foldl darstellbar ist.
- Bemerkung: die Argument von fold(l/r) braucht man nicht für den Satz, aber man kann daraus f' bestimmen (teilw. automatisch).

Ü: schwaches Inverses von `mpss`

Beweis 3. Homomorphie-Satz

Plan:

- wenn `h = foldl f e` und `h = foldr g e`, dann

  (A) $\forall x_1, y_1, x_2, y_2 : h(x_1) = h(x_2) \wedge h(y_1) = h(y_2) \Rightarrow h(x_1 +\!\!+ y_1) = h(x_2 +\!\!+ y_2)$

  Beweis: ausrechnen
- Wenn (A), dann ist h homomorph, d. h. es ex. b mit $\forall x, y : h(x +\!\!+ y) = b(h(x), h(y))$.

  Beweis: wähle ein schwaches Inverses i von h, setze $b(l, r) = h(i(l) +\!\!+ i(r))$ und ausrechnen

Beispiel: Größte Teilsumme

```
mss :: [ Int ] -> Int
mss xs = maximum $ map sum $ do
  ys <- inits xs; zs <- tails ys; return zs
```

- Darstellung als foldl/foldr?
- ... benutze `mss, mps, mps . reverse, sum`
- schwaches Inverses
- resultierende Darstellung als foldb
- Implementierung in Haskell oder Java

Beispiel: Binäre Addition

- genauer: Bestimmung des ausgehenden Übertrags bei der Addition von zwei (gleich-)langen Binärzahlen
- allgemeines Prinzip, um eine assoziative Verknüpfung zu erhalten: Funktionskomposition ist assoziativ
- im speziellen Fall betrachte

  `[(Bool,Bool)] -> ( Bool -> Bool )`

  d. h. jedem Teilsegment wird eine Funktion von Übertrag (eingehend) nach Übertrag (ausgehend) zugeordnet.
- Diese Fkt. lassen sich explizit repräsentieren (als lineare Fkt. im Halbring `(Bool,xor,and)`)
- $\Rightarrow$ Ladner-Fischer-Addierschaltkreis.

Implementierung von Folgen

als persistente Datenstruktur (in Haskell, aber nicht nur dort)

- Listen:
  - einfach verkettet: lineare Kosten
  - Cons ist lazy: Streams
- Arrays:
  - direkter Zugriff (get in $O(1)$)
  - immutable: put linear
  - append linear
- Data.Sequence: Ü: Kosten in API-Doc. nachlesen
- Data.Vector(.Unboxed):
  effiziente Kompilation durch RULES (siehe Quelltext)

Implementierung Max.-Präfixsumme

siehe git-Archiv `skpp-ws12/hom/mps-*.hs`
zusätzl. Informationen:

- Vektoren:
  `http://hackage.haskell.org/package/vector`
- effiziente Code-Erzeugung (Inlining)
  `http://article.gmane.org/gmane.comp.lang.ha`
- paper folding sequence: `http://oeis.org/A014577`

Diskussion: Kommutativität

Methode aus PLINQ
(`http://msdn.microsoft.com/en-us/library/ff9`

```
Aggregate<S, A, R>(
  this ParallelQuery<S> source,
  Func<A> seedFactory,
  Func<A, S, A> updateAccumulatorFunc,
  Func<A, A, A> combineAccumulatorsFunc,
  Func<A, R> resultSelector);
```

- in Haskell nachbauen (Typ und Implementierung)
- `combine` muß kommutativ sein
  (`http://blogs.msdn.com/b/pfxteam/archive/20`
  — warum?

Das Map/Reduce-Framework

Schema und Beispiel

```
map_reduce
  :: ( (ki, vi) -> [(ko,vm)] )  -- ^ map
  -> ( (ko, [vm]) -> vo ) -- ^ reduce
  -> [(ki,vi)] -- ^ eingabe
  -> [(ko,vo)] -- ^ ausgabe
```

Beispiel (word count)

```
ki = Dateiname, vi = Dateiinhalt
ko = Wort     , vm = vo = Anzahl
```

- parallele Berechnung von `map`
- parallele Berechnung von `reduce`
- verteiltes Dateisystem für Ein- und Ausgabe

Literatur

- Jeffrey Dean and Sanjay Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters*, OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.
  `http://labs.google.com/papers/mapreduce.htm`
- Ralf Lämmel: *Google's MapReduce programming model - Revisited*, Science of Computer Programming - SCP , vol. 70, no. 1, pp. 1-30, 2008
  `http://www.systems.ethz.ch/education/past-c`

Implementierungen

- Haskell:
  wenige Zeilen, zur Demonstration/Spezifikation
- Google:
  C++, geheim
- Hadoop:
  Java, frei (Apache-Projekt, Hauptsponsor: Yahoo)
  `http://hadoop.apache.org/`

Implementierung in Haskell

```
import qualified Data.Map as M

map_reduce :: ( Ord ki, Ord ko )
  => ( (ki, vi) -> [(ko,vm)] )  -- ^ distrib
  -> ( ko -> [vm] -> vo ) -- ^ collect
  -> M.Map ki vi -- ^ eingabe
  -> M.Map ko vo -- ^ ausgabe
map_reduce distribute collect input
    = M.mapWithKey collect
    $ M.fromListWith (++)
    $ map ( \ (ko,vm) -> (ko,[vm]) )
    $ concat $ map distribute
    $ M.toList $ input
```


Anwendung: Wörter zählen

```
main :: IO ()
main = do
  files <- getArgs
  texts <- forM files readFile
  let input = M.fromList $ zip files texts
      output = map_reduce
        ( \ (ki,vi) -> map ( \ w -> (w,1) )
                           ( words vi ) )
        ( \ ko nums -> Just ( sum nums))
        input
  print $ output
```

wo liegen die Möglichkeiten zur Parallelisierung?
(in diesem Programm nicht sichtbar.)

Hadoop

Bestandteile:

- verteiltes Dateisystem
- verteilte Map/Reduce-Implementierung

Betriebsarten:

- local-standalone (ein Prozeß)
- pseudo-distributed (mehrere Prozesse, ein Knoten)
- fully-distributed (mehrere Knoten)

Voraussetzungen:

- java
- ssh (Paßwortfreier Login zwischen Knoten)

Hadoop-Benutzung

- (lokal) konfigurieren

  `conf/{hadoop-env.sh,*-site.xml}`
- Service-Knoten starten

  `bin/start-all.sh --config /path/to/conf`
- Job starten

```
bin/hadoop --config /path/to/conf \\
    jar examples.jar terasort in out
```

Informationen:

- Dateisystem: `http://localhost:50070`,
- Jobtracker: `http://localhost:50030`

Wörter zählen

```
public static class TokenizerMapper
   extends Mapper<Object, Text, Text, IntWri
   public void map(Object key, Text value, C
public static class IntSumReducer
   extends Reducer<Text,IntWritable,Text,Int
   public void reduce(Text key, Iterable<Int
 }
public static void main(String[] args) { ...
   job.setMapperClass(TokenizerMapper.class)
   job.setCombinerClass(IntSumReducer.class)
   job.setReducerClass(IntSumReducer.class);
```

`hadoop/src/examples/org/apache/hadoop/examples/WordCo`

Sortieren

vgl. `http://sortbenchmark.org/`, Hadoop gewinnt 2008.
Beispielcode für

- Erzeugen der Testdaten
- Sortieren
- Verifizieren

(jeweils mit map/reduce)

Index-Berechnung

- Eingabe: `Map<Quelle,List<Wort>>`
- Ausgabe: `Map<Wort,List<Quelle>>`

Spezialfall: Quelle = Wort = URL, ergibt „das Web“.

Page Rank (I)

„Definition“: eine Webseite (URL) ist wichtig, wenn wichtige Seiten auf sie zeigen.

- Eingabe: Matrix `link :: (URL, URL) -> Double` mit `link(u,v)` = Wahrscheinlichkeit, daß der Besucher von u zu v geht.
- Gesucht: Vektor `w :: URL -> Double` mit `w * link = w`

Modifikationen für

- eindeutige Lösbarkeit
- effiziente Lösbarkeit

Page Rank (Eindeutigkeit)

- aus der Link-Matrix: Sackgassen entfernen (dort zufällig fortsetzen)
- diese Matrix mit völlig zufälliger Verteilung überlagern

Resultat ist (quadr.) stochastische Matrix mit positiven Einträgen, nach Satz von Perron/Frobenius

- besitzt diese einen eindeutigen größten reellen Eigenwert
- und zugehöriger Eigenvektor hat positive Einträge.

Page Rank (Berechnung)

durch wiederholte Multiplikation:
beginne mit w_0 = Gleichverteilung,
dann $w_{i+1} = L \cdot w_i$ genügend oft
(bis $|w_{i+1} - w_i| < \epsilon$)

diese Vektor/Matrix-Multiplikation kann ebenfalls mit Map/Reduce ausgeführt werden.
(Welches sind die Schlüssel?)
(Beachte: Matrix ist dünn besetzt. Warum?)

Quelle: Massimo Franceschet: *PageRank: Standing on the Shoulders of Giants* Comm. ACM 6/2011, `http://cacm.acm.org/magazines/2011/6/108660`

Übung Map/Reduce

- PLINQ/Aggregate:

  wie oft und für welche Argumente werden aufgerufen: update, combine? (abhängig von MaxDegreeOfParallelism)

- Map/Reduce (Haskell):

  Wordcount ausprobieren (Beispieltext: `http://www.gutenberg.org/etext/4300`)

  Wie bestimmt man die häufigsten Wörter? (allgemeinster Typ? Implementierung mit map/red?)

  wie multipliziert man (dünn besetze) Matrizen mit map/red?

- Map/Reduce mit Hadoop:

  `http://www.michael-noll.com/tutorials/runni`

Ausblick, Zusammenfassung

Ausblick: Hochparallele preiswerte Hardware

Grafikkarten mit z. B. 500 Rechenkernen (CUDA, OpenCL)
Eigenschaften:

- Verarbeitungsmodell ist low-level (C), aber (in Grenzen) hardwareunabhängig
- Rechnungen (im Kern) sehr schnell
- schneller Speicher (je Kern) sehr klein

vlg. Übersichtsvortrag
`http://www.imn.htwk-leipzig.de/~waldmann/tal`

Komplexitätstheorie

. . . für parallele Algorithmen
Klassen:

- NC $=$ polylogarithmische Zeit, polynomielle Anzahl von Prozessoren
- P $=$ polynomielle Zeit
- NC $\subseteq$ P

Reduktionen:

- $\le_L$ logspace-Reduktion, Eigenschaften
- P-vollständige Probleme (Bsp: Tiefensuchreihenfolge)

(vgl. für sequentielle Algorithmen: Klasse NP, Polynomialzeitreduktion $\le_P$, NP-Vollständigkeit)

Zusammenfassung

- Temporal-Logik (zur Beschreibung von Eigenschaften nebenläufiger Systeme)
- sicherer Zugriff auf gemeinsamen Speicher:

  software transactional memory (STM, TVar, . . .)
- Prozesse mit lokalem Speicher: CSP, Kanäle, Agenten; verteilte Programme
- parallele Auswertung funktionaler Programme (Strategie-Annotationen)
- balancierte folds, map/reduce

Beispielklausur:
`http://www.imn.htwk-leipzig.de/~waldmann/edu`