

Sprachkonzepte  
der Parallelen Programmierung  
Vorlesung  
SS 11, WS 12, SS 13, WS 15, WS 16

Johannes Waldmann, HTWK Leipzig

23. Januar 2017

# Sprachkonzepte der parallelen Programmierung

- ▶ programming language concepts for concurrent, distributed, and parallel computing
- ▶ why? 1. application requires it, 2. hardware allows it
- ▶ optional course for BSc. computer science students, in their 5th semester (of 6)
- ▶ each week (of 14): one lecture, one lab class (discussing homework, programming exercises)
- ▶ finally, written exams (closed book) 120 min

# Concepts of Parallelism

- ▶ non-deterministic
  - ▶ concurrent (nebenläufig)  
interleaved execution of components
  - ▶ distributed (verteilt)  
as above, plus: explicit data transfer (messages)

e.g., responsive multi-user system requires concurrency  
(can be simulated on sequential hardware: OS)
- ▶ deterministic  
for application probl. without concurrency requirement,  
use hardware parallelism to solve them faster  
e.g., matrix multiplication, text/image analysis

# From Concept to Implementation

notice the gap:

- ▶ quite often, we want deterministic parallelism
- ▶ but hardware is concurrent (e.g., several cores) and distributed (e.g., per-core memory/cache)

who bridges the gap?

- ▶ WRONG: the *programmer* (application program handles sequencing, synchronisation and messaging)
- ▶ RIGHT: the *language* (with libraries, compiler, run-time system) (program expresses intent)

note the difference between:  $\sum_{0 \leq i < n} x_i$  (intent)

and: `for (i=0; i<n; i++) { s+=x[i]; }` (sequencing)

# Abstract! Abstract! Abstract!

main thesis

- ▶ *higher* abstraction level of the language
- ▶  $\Rightarrow$  *easier* for compiler and RTS to use hardware specifics (e.g., parallelism) for efficient execution

example (C#, mono) just one annotation expresses the intent of parallel execution:

```
Enumerable.Range(0, 1<<25)
    .Select(bitcount).Sum()
Enumerable.Range(0, 1<<25).AsParallel()
    .Select(bitcount).Sum()
```

this is why we focus on functional programming  
(e.g., `Select` is a higher-order function)

## Why did this work, exactly?

```
Enumerable.Range(...).AsParallel().Sum()
```

- ▶ **technically**, `AsParallel()` produces `ParallelQuery<T>` from `IEnumerable<T>`, and `Sum()` has a clever implementation for that type
- ▶ mathematically (“clever”), addition is *associative*, so we can group partial sums as needed
- ▶ if an operation is not associative?  
e.g., the final carry bit in addition of bitvectors  
then we should find a modification that is!  
because that allows for *straightforward*  
and *adaptable* (e.g., to number of cores) parallelism

# Types for Pure Computations

measure run-times and explain (C# vs. Haskell)

```
Enumerable.Range(0,1<<25).Select(bitcount).Sum()
Enumerable.Range(0,1<<25).Select(bitcount).Count()
Enumerable.Range(0,1<<25)                          .Count()
length $ map bitcount $ take (2^25) [ 0 .. ]
length           $ take (2^25) [ 0 .. ]
```

- ▶ elements of list are not needed for counting
- ▶ computation of elements cannot be observed  
it has not side effects (Nebenwirkung)  
(this follows from `bitcount :: Int -> Int`)  
the Haskell RTS never calls `bitcount`,
- ▶ the C# type `int->int` includes side effects,  
so the RTS must call the function.

# If we absolutely must program imperatively,

(imp. program execution = sequence of state changes)

- ▶ then we are on dangerous ground already for the sequential case  
proving an imperative program correct requires complicated machinery (Hoare calculus)  
hence it is often not done  
(for functional programs just use equational reasoning)
- ▶ we need even more caution (and discipline) for concurrent imperative programs  
need concurrency primitives
  - ▶ that have clear semantics
  - ▶ that solve typical problems
  - ▶ that are composable (to larger programs)



# Typical Concurrency Problems

- ▶ mutual exclusion
  - at most one process gets to access a shared resource (e.g., a shared memory location)
- ▶ producers and consumers, readers and writers
  - ▶ cannot consume item before it is produced,
  - ▶ cannot consume item twice
- ▶ concurrent mutable data structures
  - ▶ counters
  - ▶ collections (hash maps, ...)

# Semantics for Concurrent Systems

... via mathematical models:

- ▶ Petri nets (Carl Adam Petri, 1926–2010)  
(automata with distributed state)
- ▶ process algebra (Bergstra and Klop 1982, Hoare 1985)  
(regular process expressions and rewrite rules)  
`http://theory.stanford.edu/~rvg/process.html`
- ▶ modal logic  
(statements about time-dependent properties)  
application: model checking,  
e.g., SPIN `http://spinroot.com/`

# Concurrency Primitives

- ▶ locks (Semaphores, E.W. Dijkstra 1974) <http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD74.PDF>  
of historical importance, but ... *locks are bad*  
(in particular, not composable)
- ▶ no locks  
atomic, non-blocking (“optimistic”) execution of
  - ▶ elementary operations (compare-and-swap)  
realized in hardware  
<http://stackoverflow.com/questions/151783/>
  - ▶ transactions (STM, software transactional memory)  
<http://research.microsoft.com/en-us/um/people/simonpj/papers/stm/> in Haskell, Clojure

# Homework

1. which are associative? (give proof or counter-example)
  - 1.1 on  $\mathbb{Z}$ : multiplication, subtraction, exponentiation
  - 1.2 on  $\mathbb{B}$  (booleans): equivalence, antivalence, implication
  - 1.3 on  $\mathbb{N}^2$ :  $(a, b) \circ (c, d) := (a \cdot c, a \cdot d + b)$
2. sum-of-bitcounts
  - 2.1 re-do the C# bitcounting example in Java (hint:  
`java.util.stream`)
  - 2.2 discuss efficient implementation of  
`int bitcount (int x);` (hint: time/space trade-off)
  - 2.3 discuss efficient implementation of sum-of-bitcounts
    - 2.3.1 from 0 to  $2^e - 1$
    - 2.3.2 bonus: from 0 to  $n - 1$  (arbitrary  $n$ )  
hint:  
how did little Carl Friedrich Gauß do the addition?  
morale:  
the computation in the example should never be done in real life, but it makes a perfect test-case since it keeps the CPU busy and we easily know the result.
3. sorting network exercise: <https://autotool.imn.htwk-leipzig.de/new/aufgabe/2453>
4. on your computer, install compilers/BTS for languages:

# Einleitung

- ▶ Verhalten nebenläufiger Systeme *spezifizieren* und *modellieren*
- ▶ Spezifikation (Beispiel): Spursprache (Menge der möglichen Reihenfolgen von atomaren Aktionen)
- ▶ Modell (Beispiel): Petri-Netz (nebenläufiger Automat) eingeführt von Carl Adam Petri, 1962

Vergleiche: Beschreibung/Modellierung sequentieller Systeme durch reguläre Sprachen/endliche Automaten

Vorgehen hier: erst konkrete Modelle, dann Spezifikationssprache (Logik).

# Definition: Netz

Stellen/Transitions-Netz  $N = (S, T, F)$

- ▶  $S$  eine Menge von *Stellen*
- ▶  $T$  eine Menge von *Transitionen*,  $S \cap T = \emptyset$
- ▶  $F \subseteq (S \times T) \cup (T \times S)$  eine Menge von *Kanten*

das ist ein gerichteter bipartiter Graph

Bezeichnungen:

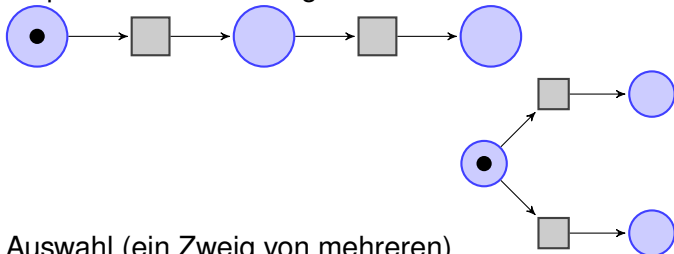
- ▶ Vorbereich (Eingänge) einer Transition:  
 $\text{Vor}_N(t) = \{s \mid (s, t) \in F\}$
- ▶ Nachbereich (Ausgänge) einer Transition:  
 $\text{Nach}_N(t) = \{s \mid (t, s) \in F\}$ .

# Zustände, Übergänge

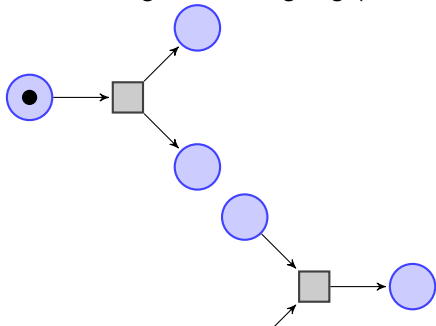
- ▶ *Zustand* eines Netzes  $N$  ist Abbildung  $z : S \rightarrow \mathbb{N}$   
(für jede Stelle eine Anzahl von Marken)
- ▶ in Zustand  $z$  ist eine Transition  $t$  *aktiviert*,  
wenn jede Stelle ihres Vorbereiches  
wenigstens eine Marke enthält:  $\forall s \in \text{Vor}(t) : z(s) \geq 1$
- ▶ eine aktivierte Transition *schaltet*: verbraucht Marken im  
Vorbereich, erzeugt Marken im Nachbereich.
- ▶ Bezeichnung  $z_1 \xrightarrow{t} z_2$ :  
aus Zustand  $z_1$  entsteht durch Schalten von  $t$  der Zust.  $z_2$ .
- ▶ Def.  $z_1 \xrightarrow{t} z_2$ : erfordert 4 Fälle: alle Kombinationen  
von:  $s \in \text{Vor}(t)$ ,  $s \notin \text{Vor}(t)$  mit  $s \in \text{Nach}(t)$ ,  $s \notin \text{Nach}(t)$   
effizientere Notation in Modell mit Kantengewichten.

# Petri-Netze modellieren...

- ▶ sequentielle Ausführung



- ▶ Auswahl (ein Zweig von mehreren)
- ▶ nebenläufige Verzweigung (mehrere Zweige)







# Petri-Netze und UML

UML-2.5, Sect. 13.2.1

A variety of behavioral specification mechanisms are supported by UML, including:

- ▶ StateMachines that model finite automata (see C. 14)
- ▶ Activities defined using Petri-net-like graphs (see C. 15)
- ▶ Interactions that model partially-ordered sequences of event occurrences (see C. 17).

# Sprache eines Netzes

- ▶ abstrahiere von Folge von Zuständen  $z_0 \rightarrow_{t_1} z_1 \rightarrow_{t_2} z_2 \dots$   
zur *Spur*: Folge der dabei geschalteten Transitionen  $t_1 t_2 \dots$   
ist Wort über Alphabet = Transitionen
- ▶ für gegebenes Netz und Startzustand: betrachte Menge  
aller möglichen Spuren (Spursprache)  
vergleiche: Sprache eines endlichen Automaten
- ▶ aber: es gibt Petri-Netze mit komplizierten (= nicht  
regulären) Spursprachen

# Kapazitäten und -Schranken

Erweiterung:

- ▶ jede Kante bekommt eine *Gewicht* (eine positive Zahl), beschreibt die Anzahl der Marken, die bei jedem Schalten durch die Kante fließen sollen.

Einschränkung:

- ▶ Stellen können einer *Kapazität* bekommen (eine positive Zahl), beschreibt die maximal erlaubte Anzahl von Marken in dieser Stelle

falls alle Kapazitäten beschränkt  $\Rightarrow$  Zustandsmenge endlich  
(aber mglw. groß)  $\Rightarrow$  vollständige Analyse des  
Zustandsübergangsgraphen (prinzipiell) möglich

# Formale Definition der $\ddot{U}$ -Relation

- ▶ Netz mit Kantengewichten:  $F : (S \times T) \cup (T \times S) \rightarrow \mathbb{N}$
- ▶ Beziehung zu einfachem Modell:  
keine Kante: Gewicht 0, einfache Kante: Gewicht 1
- ▶ Transition  $t$  ist in Zustand  $z$  aktiviert:  $\forall s : z(s) \geq F(s, t)$ .
- ▶ Zustandsübergang:  $z_1 \xrightarrow{t} z_2$ :  
 $\forall s : z_1(s) - F(s, t) + F(t, s) = z_2(s)$
- ▶ beachte: durch *Verallgemeinerung* des Modells  
wird Notation hier *einfacher* ...  
und damit auch Beweise, die Notation benutzen.

# Bedingung/Ereignis-Netze

... erhält man aus allgemeinem Modell durch:

- ▶ jede Kante hat Gewicht 1
- ▶ jede Kapazität ist 1  
(dadurch wird der Zustandsraum endlich!)

Beispiele:

- ▶ Ampelkreuzung  
(zwei Ampeln grün/gelb/rot, nicht gleichzeitig grün)
- ▶ speisende Philosophen
- ▶ Definition und Analyse von Lebendigkeit, Fairness

# Eigenschaften von Petri-Netzen

Definitionen (für Netz  $N$  mit  $d$  Stellen, Zustand  $m \in \mathbb{N}^d$ )

- ▶  $M \subseteq \mathbb{N}^d$ : Nachfolger  $\text{Post}_N(M) = \{y \mid m \in M, m \rightarrow_N y\}$
- ▶ Mehr-Schritt-Nachfolger:  $\text{Post}_N^*(M)$

Eigenschaften (Beispiele):

- ▶ Erreichbarkeit: gilt  $m_0 \rightarrow_N^* m_1$ ?
- ▶ Beschränktheit: ist  $\text{Post}_N^*(m_0)$  endlich?
- ▶ Platz-Beschränktheit:  $\{m(p) \mid m \in \text{Post}_N^*(m_0)\}$  endlich?

Alain Finkel und Jerome Leroux: *Neue, einfache Algorithmen für Petri-Netze*, Informatik-Spektrum 3/2014, S. 229–236

# Beschränktheit ist entscheidbar

$\text{Post}_N^*(m_0)$  endlich?

Entscheidungsverfahren: wir zählen abwechselnd auf:

- ▶ A: Elemente von  $\text{Post}^*(m_0)$  (z.B. Breitensuche)
- ▶ B: Kandidaten für Zeugen für Unbeschränktheit:

Kandidat ist  $(s, t) \in T^* \times T^+$ ,

ist Zeuge, wenn  $m_0 \rightarrow^s x \rightarrow^t y$  mit  $x \leq y$  und  $x \neq y$

zu zeigen ist:  $\text{Post}_N^*(m_0)$  unendlich  $\iff$  Zeuge existiert

„ $\Leftarrow$ “: ist klar. Für „ $\Rightarrow$ “:

- ▶  $\rightarrow$  auf  $\text{Post}^*(m_0)$  ist unendlichen Baum endlichen Grades
- ▶ enthält unendlichen Pfad (Lemma von König)
- ▶ dieser Pfad enthält passendes  $(x, y)$  (Lemma v. Higman)



# Lemma von Higman, WQO

- ▶ Def. eine Relation  $\leq$  auf  $M$  heißt *WQO* (*wohl-quasi-Ordnung*), falls gilt: es gibt keine unendliche  $\leq$ -Antikette.
- ▶ Def. eine Menge  $A \subseteq M$  heißt *Antikette*, falls  $\forall x, y \in A : x \neq y \Rightarrow x \not\leq y$ .  
(Def. ... *Kette*, falls  $\dots \Rightarrow x \leq y \vee y \leq x$ .)
- ▶ Bsp:  $(\mathbb{N}, \leq)$  ist WQO.  $(\mathbb{N}, |)$  (Teilbarkeit) ist ?
- ▶ Versionen von Higmans Lemma:  
Satz:  $(\mathbb{N}^d, \leq)$  (komponentenweise  $\leq$ ) ist WQO.  
Satz:  $(\Sigma^*, \sqsubseteq)$  ist WQO, wobei  $(u \sqsubseteq v) : \iff$   
 $u$  entsteht aus  $v$  durch Löschen von Buchstaben

# Aufgaben

- ▶ Vergleich Wechselschalter (Kastens/Kleine Büning Abb. 7.23) mit Netz aus Vorlesung
- ▶ Zustandsgraph (K/KB Aufg. 7.16)
- ▶ autotool-Aufgaben (reachability, deadlock)
- ▶ zu Petri-Netz für gegens. Ausschluß (Ampelsteuerung):  
formuliere eine Verschärfung der angegebenen Invariante, die man durch Induktion über Länge der Schaltfolge beweisen kann.
- ▶ Beispiele Petri-Netze zur Modellierung (Modell)eisenbahn:
  - ▶ Zug darf Streckenabschnitt nur befahren, wenn Signal grün zeigt
  - ▶ Zug darf Weiche nicht aus dem falschen Zweig befahren
- ▶ XOR-Verzweigung (mit späterer Zusammenführung) durch Petri-Netz ist einfach. Wie geht das für OR?  
(ein Zweig *oder beide* Zweige werden ausgeführt)

# 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

# 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/ccp/](http://se.inf.ethz.ch/courses/2012a_spring/ccp/)
- ▶ Sibylle Schwarz: Verifikations- und Spezifikationsmethoden (Abschnitt 3: Model Checking) <http://whz-cms-10.zw.fh-zwickau.de/sibsc/lehre/ws11/veri/>

# 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  $b : S \rightarrow (V \rightarrow \mathbb{B})$   
d.h.,  $b(s)$  ist eine Belegung der Variablen  $V$

hier speziell:

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

Beispiel:

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

# Omega-Wörter und -Sprachen

- ▶ jede lineare Kripke-Struktur über  $V$  entspricht einem unendlichen Wort über  $\Sigma = 2^V$   
Bsp:  $(0, 1)(0, 0)(0, 1)(1, 0)(1, 1)(1, 0)(1, 1) \dots$
- ▶ ein unendliches Wort (Omega-Wort) über  $\Sigma$  ist eine Abbildung  $\mathbb{N} \rightarrow \Sigma$
- ▶  $\Sigma^\omega$  bezeichnet die Menge aller Omega-Wörter über  $\Sigma$
- ▶ Schreibweise für Omega-Wörter mit schließlich periodischer Struktur:  
 $(0, 1)(0, 0)(0, 1) ((1, 0)(1, 1))^\omega$   
vgl. unendl. Dezimalbrüche  $3/22 = 0.1\overline{36}$

# PLTL: propositional linear time logic

Syntax:

- ▶ Variablen  $p, q, \dots$ , logische Operatoren  $\neg, \vee, \wedge, \Rightarrow, \dots$
- ▶ temporale Operatoren: immer  $\square$ , irgendwann  $\diamond, \dots$

Beispiele:  $\diamond(p \vee q), \square\diamond p, \diamond\square p$

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

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

Übung:  $\diamond\square\phi \Rightarrow \square\diamond\phi$  ist allgemeingültig (gilt in jeder Struktur),  
... aber die Umkehrung nicht

# PLTL-Spezifikationen von Systemeigenschaften

- ▶ gegenseitiger Ausschluß (*mutual exclusion*):  
Variablen:  $p_i :=$  Prozeß  $i$  besitzt eine Ressource
  - ▶ Spezifikation (2 Prozesse):  $\Box \neg(p_1 \wedge p_2)$
  - ▶ Übung: für 3 Prozesse lautet die Formel nicht  $\Box \neg(p_1 \wedge p_2 \wedge p_3)$ . Warum nicht? Wie dann?
- ▶ Fairness (kein Verhungern, *no starvation*)  
Variablen:  $A_i :=$  Prozeß  $i$  beantragt Ressource;  $P_i$   
Spezifikation:  $\Box(A_1 \Rightarrow \Diamond P_1) \wedge \dots \wedge \Box(A_n \Rightarrow \Diamond P_n)$



# 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  $1 = \text{wert}(F, w, 0)$
- ▶ Erfüllbarkeits-Problem:
  - ▶ Eingabe: eine PLTL-Formel  $F$
  - ▶ Frage: gibt es  $w \in \Sigma^\omega$  mit  $1 = \text{wert}(F, w, 0)$

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

## $\omega$ -(reguläre) Sprachen

- ▶ Alphabet  $\Sigma$ ,
- ▶  $\omega$ -Wort  $w \in \Sigma^\omega$ : Abbildung  $\mathbb{N} \rightarrow \Sigma$
- ▶  $\omega$ -Sprache  $L \subseteq \Sigma^\omega$ : Menge von  $\omega$ -Wörtern
  
- ▶  $\omega$ -reguläres Wort: hat die Form  $u \cdot v^\omega$  mit  $v \neq \epsilon$ .

Achtung: es gibt kein Produkt von  $\omega$ -Wörtern,  
also auch keine geschachtelten Omegas.

- ▶  $\omega$ -reguläre Sprache:

beschrieben durch  $\omega$ -regulären Ausdruck:

$$P_1 \cdot K_1^\omega \cup \dots \cup P_n \cdot K_n^\omega \quad \text{mit } P_i, K_i \text{ regulär und } \neq \emptyset, \epsilon \notin K_i$$

Achtung: eine  $\omega$ -reguläre Sprache (Bsp.  $(a + b)^\omega$ )  
kann auch nicht-reguläre  $\omega$ -Wörter enthalten.

# Übung PLTL

- ▶ Ist die Struktur  $0(001)^\omega$  ein Modell der Formel  $\diamond\Box p$  ?
  - ▶ Gibt es ein Modell für  $\Box(p \iff \diamond\neg p)$ ?
  - ▶ Formalisieren Sie (mit den Variablen  $p_i$  für „Prozeß  $i$  besitzt Ressource“)
    - ▶  $F$  = Prozeß 1 besitzt die Ressource unendlich oft,
    - ▶  $G$  = Prozesse 1 und 2 besitzen die Ressource nie gleichzeitig,
    - ▶  $H$  = immer, wenn Prozeß 1 die Ressource besitzt, dann besitzt Prozeß 2 diese nicht, wird sie aber später erhalten.
  - ▶ Für alle 8 Konjunktionen von  $\{F, G, H, \neg F, \neg G, \neg H\}$ : geben Sie jeweils ein Modell als  $\omega$ -reguläres Wort an (falls möglich).
  - ▶ durch die Operatoren
    - ▶  $FUG$ :  
Es gibt einen Zeitpunkt, zu dem  $G$  gilt. Bis dahin gilt  $F$ .
    - ▶  $XF$ : im nächsten Zeitpunkt gilt  $F$ .
- wird PLTL erweitert.
- ▶ Gegeben Sie die formale Semantik von  $U$  und  $X$  an.
  - ▶ Wie kann man  $\diamond$  durch Until realisieren?
  - ▶ Realisieren Sie Until durch Next (mit einer Hilfsvariablen  $p_i$

# Vergleich der PLTL-Operatoren

Syntax:

deutsch	Symbol	englisch	autotool	Spin
irgendwann	$\diamond$	finally (eventually)	F	$\langle \rangle$
immer	$\square$	globally, generally	G	[ ]
bis		until	U	U
nächster		next	X	X

Ausdrucksstärke von PLTL( $M$ ) für  $M \subseteq \{F, G, U, X\}$ :

- ▶ PLTL(F, G) kann  $(abc)^\omega$  nicht von  $(acb)^\omega$  unterscheiden dabei ist  $a$  Abkürzung für  $\{a = 1, b = 0, c = 0\}$ , usw.
- ▶ ... aber PLTL(U), durch  $(aU b) U c$
- ▶ PLTL(F, G, U) kann  $(ab)^\omega$  nicht von  $(abb)^\omega$  unterscheiden,
- ▶ ... aber PLTL(X), durch ... ?

# Überblick

- ▶ <http://spinroot.com/spin/whatispin.html>
- ▶ Eingabe:
  - ▶ nebenläufiges System (als Programmtext in der Sprache Promela = process meta language)
  - ▶ Spezifikation (im Program: `assert`, separat: LTL-Formel)
- ▶ SPIN kann
  - ▶ System simulieren (zufällig, benutzergesteuert)
  - ▶ Erfüllung der Spezifikation beweisen
  - ▶ für Nicht-Erfüllung einen Beweis (Spur) berechnen

# Promela

- ▶ Prozesse: `active[2] proctype p() {...}`
- ▶ Datentypen: `bool, byte, int, Arrays`
- ▶ Ausdrücke: ähnlich C, aber *nebenwirkungsfrei*
- ▶ Programmablaufsteuerung:
  - ▶ goto/label, Blöcke, Schleifen: wie in C
  - ▶ boolescher Ausdruck als blockierender Test
  - ▶ atomare Ausführung: `atomic { ... }`
  - ▶ nichtdeterministische Auswahl: `if :: s1 :: s2 fi`  
Wächter-Bedingungen, Blockierung, `else`
  - ▶ wiederholte n.-d. Auswahl: `do :: s1 :: s2 od`  
Wächter-Bedingungen, `break`

<http://spinroot.com/spin/Man/Quick.html>

# Einfaches SPIN-Beispiel

```
byte n = 0, finished = 0;
active [2] proctype P() {
  byte a, c = 0;
  do :: c == 10 -> break
    :: else -> a = n; a++; n = a; c++
  od;
  finished++
}
ltl { [] (finished == 2 -> n >= 10) }
```

- ▶ ispin bsp.pml, **Edit/View**
- ▶ **Simulate/Replay** (Random, with seed: ...)
- ▶ **Verification** (Never Claims, use claim)
- ▶ danach Sim/Replay (Guided, with trail)

# Simulation/Analyse von Petrinetzen

## Bsp. Ampelschaltung

```
byte l1=0, l2=1, l3=0, m=1, r1=0, r2=0, r3=0;
active[1] proctype net() {
  do
    :: atomic { l2>0 ; l2--; l3++ }
    :: atomic { l3>0 && m>0; l3--; m--; l1++ }
    ...
  od
}
```

- ▶ vervollständige das Programm, simuliere,
- ▶ betrachte Zustandsraum (Edit/View: Automata)
- ▶ formuliere gegenseitigen Ausschluß als LTL, verifiziere
- ▶ ändere Netz oder Spezifikation, verifiziere/falsifiziere



# Entscheiden der PLTL-Erfüllbarkeit mit Spin

- ▶ Beispiel: „gibt es ein Modell für  $G(p \iff F \neg p)$ “?
- ▶ mechanische Lösung:

```
bool p;  
active proctype A ()  
  { do :: p = 0; :: p = 1; od }  
ltl { ! always (p <-> eventually ! p) }
```

Verifikation mit Spin liefert *no errors found*

- ▶ Antwort: Formel ist nicht erfüllbar
- ▶ Begründung:
  - ▶ Spursprache von Prozeß  $A$  ist  $\{0, 1\}^\omega$  (d.h. *alle* Wörter), wegen nichtdeterministischer Auswahl in jedem Schritt
  - ▶ der LTL-*claim* ist die Negation der Formel  $\phi$  aus der Aufgabenstellung. Spin beweist, daß diese in jeder Spur wahr ist, also ist  $\phi$  für jede Spur falsch, also besitzt  $\phi$  kein Modell.
- ▶ *Vorsicht*: Variable  $p$  wird leider automatisch mit 0 initialisiert, d.h. die Spursprache ist  $0\{0, 1\}^\omega$ , um das zu umgehen, muß man noch ein X einfügen (?)

# Threads erzeugen und starten

Thread-Objekt implementiert `run()`,  
diese Methode wird aufgerufen durch `start()`,  
das aber sofort zurückkehrt (mglw. bevor `run()` endet).

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

alternative Notation (Java  $\geq$  8)

```
new Thread( () -> System.out.println(t) );
```

## Auf das Ende von Threads warten

`t.join()` blockiert aufrufenden Thread  
und kehrt erst zurück, wenn `t` beendet ist:

```
t1.start(); t2.start();  
...  
t1.join() ; t2.join();
```

- ▶ das ist die einfachste Möglichkeit der Synchronisation, benutzt nur Threads selbst
- ▶ es gibt viele weitere Möglichkeiten, diese benutzen zusätzliche Objekte (Sperrern)

# Gemeinsamer Speicher

(Vorsicht, Code ist absichtlich falsch)

```
int s = 0; // gemeinsamer Speicher
// Threads erzeugen:
for (int t=0; t<threads; t++) {
    new Thread() ( () ->
        for (int i = 0; i<steps; i++) s++;
// Threads starten: ...
// auf Threads warten: ...
System.out.println (s);
```

- ▶ Ausgabe ist i.A. deutlich kleiner als `threads*steps`.
- ▶ `s++` ist nicht atomar
- ▶ tatsächlich wird noch viel weniger garantiert

# Das Java-Speichermodell

- ▶ beschreibt garantiertes Verhalten beim Zugriff nebenläufiger Programme auf gemeinsamen Speicher (Objekt-Attribute, Klassen-Attribute, Array-Inhalte)
- ▶ Definition: JLS Kap. 17.4  
<http://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.4>
- ▶ Erläuterungen: William Pugh: <http://www.cs.umd.edu/~pugh/java/memoryModel/>

# Sequentielle Konsistenz (Plan)

- ▶ moderne Compiler und Hardware (Prozessoren/Caches) können elementare Anweisungen umordnen und verändern,
- ▶ wenn trotzdem das Ergebnis berechnet wird, das der Programmierer erwartet, nämlich das *ohne Umordnung*
- ▶ falls das so ist, heißt der Compiler *sequentiell konsistent*
- ▶ sequentielle Konsistenz gilt in Java innerhalb jedes Threads, aber *nicht* zwischen Threads
- ▶ das Erreichen der gewünschten Konsistenz bleibt dem Programmierer überlassen  
(zu erreichen durch Sprache und Bibliotheken)

# Beispiel Umordnung

```
vorher : A == 0; B == 0;  
Thread 1: r2 = A ; B = 1;  
Thread 2: r1 = B ; A = 2;
```

- ▶ Ist schließlich `r1 == 1; r2 == 2` möglich?
- ▶ in SPIN nicht (Ü: ausprobieren)
- ▶ in Java doch, denn ...

# Beispiel Code-Änderung

vorher: `p == q` und `p.x == 0`

Thread 2:

```
r6=p; r6.x=3;
```

Thread 1:

```
r1=p; r2=r1.x; r3=q; r4=r3.x; r5=r1.x;
```

- ▶ in `r2, r4, r5` wird gleiche Stelle gelesen, wir erwarten Resultate `0, 0, 0` oder `0, 0, 3` oder `0, 3, 3` oder `3, 3, 3`

- ▶ transformiere Code in Thread 1 zu

```
r1=p; r2=r1.x; r3=q; r4=r3.x; r5=r2;
```

dann ist Resultat `0, 3, 0` möglich.



# Def. Speichermodell (Plan)

Merksatz:

- ▶ in sequentiellen Java-Programmen bestimmt die Programm-Ordnung die Ausführungsordnung beides sind totale Ordnungen
- ▶ in nebenläufigen Java-Programmen *gibt es keine totale Ausführungsordnung*  
... das Speichermodell definiert nur eine *partielle* „happens-before“-Halbordnung

## Def. Speichermodell (Detail)

- ▶ (JLS 17.4.3) Programm-Ordnung (P.O.)  
(total je Thread, Quelltextreihenfolge der Aktionen)
- ▶ (.4) Synchronisations-Ordnung (S.O.)  
(total, auf bestimmten Aktionen: volatile-write/read, Thread-start, Beginn, Ende, join)
- ▶ (.4) *synchronises-with* (S.W.) (partiell)  
Schreiben auf volatile-Variable  $v$  S.W. jedem folgenden (bzgl. S.O.) Lesen von  $v$
- ▶ (.5) *happens-before* (vereinfacht): (H.B.)  
transitive Hülle von  $(P.O. \cup S.W.)$
- ▶ jede well-formed (.7) und kausale (.8) Ausführung ist erlaubt.

# Data Races

- ▶ Def: ein *data race* ist eine Menge von Zugriffen auf eine gemeinsame Variable, die durch *happens-before* nicht total geordnet ist.
- ▶ Def: ein *korrekt synchronisiertes* Programm ist ein Programm ohne *data races*.

# volatile-Variablen

vereinfachtes Denkmodell

(= Veranschaulichung der happens-before-Relation)

- ▶ jeder Thread hat eine Kopie (einen Cache) der gemeinsamen Variablen (des Heaps)
- ▶ Synchronisation nur bei bestimmten Aktionen, u.a.

Benutzen von `volatile` deklarierten Variablen:

- ▶ beim Lesen ein *refresh* (Heap → Cache)
- ▶ beim Schreiben ein *flush* (Cache → Heap)

Thread-Verwaltung:

- ▶ start: refresh
- ▶ join: flush

# Übung JMM

- ▶ Beispiele für totale und nicht totale Halbordnungen  
Präfix-Ordnung auf Wörtern, lexikografische O. auf Wörtern, lex. O. auf Zahlenpaaren, komponentenweise O. auf Zahlenpaaren,  
Def. komponentenweises Produkt von Relationen, lex. Produkt von Relationen.
- ▶ Streams in Java 8  
Erzeugung (aus anderen Typen), Transformation, Konsumtion (zu anderen Typen)
- ▶ Funktionale Schnittstellen, Methoden-Verweise und Lambda-Ausdrücke in Java 8  
Notation, Benutzung, Typinferenz
- ▶ Beispiel aus Goetz: JCP, Abschn. 3.2

```
volatile
    boolean running = true;
Thread 1:
    while (running) { }
Thread 2:
```

# Semaphore

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

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

Invariante:  $S = S_0 + \#\text{Signal} - \#\text{Wait}$

( $\#\text{Wait}$  = Anzahl der *abgeschlossenen* Aufrufe von  $\#\text{Wait}$ , entspr. für  $\#\text{Signal}$ )

Beweis der Invarianz: Induktionsanfang und 4 Fälle im Induktionsschritt

# Semaphor: Geschichte

- ▶ E. W. Dijkstra: *Cooperating Sequential Processes*, 4. *The General Semaphore*, TU Eindhoven 1965  
<http://www.cs.utexas.edu/~EWD/transcriptions/EWD01xx/EWD123.html>
- ▶ J. B. Calvert: *The Origin of the Railway Semaphore (Its evolution from the optical telegraph)*  
<http://mysite.du.edu/~jcalvert/railway/semaphor/semhist.htm>
- ▶ Monty Python: *Semaphore Version of Wuthering Heights*

# Gegenseitiger Ausschluß (grundsätzlich)

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

## Eigenschaften:

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



# Gegenseitiger Ausschluß (Korrektheit)

Bezeichnungen:

- ▶  $S$ : Zustand des Semaphors
- ▶  $C$ : Anzahl der Prozesse in kritischem Abschnitt

Zeige Invariante:  $S + C = 1$ .

Beweis:

- ▶  $C = \#Wait - \#Signal$  lt. Programmtext
- ▶  $S = 1 + \#Signal - \#Wait$  lt. Semaphor-Invariante

aus Invariante folgt Korrektheit ( $C \leq 1$ )

# Gegenseitiger Ausschluß in Spin

```
byte s = 1;  active [2] proctype P () {
  do :: skip ; R : atomic { s > 0; s-- }
    C : skip;
    s++;
  od }
```

formuliere und prüfe Bedingungen:

- ▶ Korrektheit (gegenseitiger Ausschluß)

`always ! ( P[0]@C && P[1]@C )`

- ▶ Fairness? — Nein. Deswegen ist das keine korrekte Simulation eines Semaphors.

`always ( P[0]@R -> eventually P[0]@C )`

- ▶ Progress?

# Semaphore und Monitore

- ▶ Semaphore (ist Objekt)  
Aufgabe ist Synchronisation von *Threads*
- ▶ Monitor (ist Menge von Methoden)  
Aufgabe ist Zugriffskontrolle für Daten
- ▶ Monitor kann durch Semaphor realisiert werden:  
jeder Zugriff (jede Methode) muß Semaphor erwerben und wieder abgeben
- ▶ einfache Notation in Java durch `synchronized`

# Monitore in Java

- ▶ die `synchronized`-Methoden einer Klasse `C` bilden für jedes Objekt von `C` einen Monitor (für jedes Objekt von `C` kann jederzeit höchstens eine Monitor-Methode laufen)
- ▶ diese Methoden sind *re-entrant*: während man eine Monitor-Methode ausführt, kann man weitere Methoden des gleichen Monitors aufrufen (deswegen funktioniert Implementierung mit Semaphor doch nicht — diese würde verklemmen)
- ▶ Spezialfall: ein-elementiger Monitor: Code-Block

```
Object lock = new Object ();  
synchronized (lock) { ... }
```

## Explizites wait/notify für Monitore

- ▶ durch `synchronized` sind Erwerb und Freigabe der Sperre versteckt und automatisch richtig geschachtelt
- ▶ explizites Sperren und Freigeben sind auch möglich:  
Methoden `wait`, `notify`, `notifyAll`:

```
synchronized void take (..) {  
    while (taken) this.wait (); taken = true; }  
synchronized void drop (..) {  
    taken = false; this.notifyAll (); }
```

- ▶ Benutzung nur innerhalb eines Monitors,
- ▶ `wait()` in Schleife, siehe *spurious wake-ups* in <http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#wait-->

# 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) { this.wait (); }  
        taken = true;    }  
    synchronized void drop () {  
        taken = false; this.notifyAll ();    } }  
}
```

Q: warum wird expliziter Semaphor (wait/notify) benutzt?

A: jeder Prozeß (Philosoph) benötigt zwei Ressourcen (Gabeln) gleichzeitig, kann aber nicht zwei `synchronized`-Methoden gleichzeitig ausführen.

## 5 Philosophen

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

welche Eigenschaften? wie kann man das ggf. reparieren?

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

**Quelltexte:** <https://gitlab.imn.htwk-leipzig.de/waldmann/skpp-ws16>



# Übung Monitor

Verhalten dieses Programmes ausprobieren, diskutieren:

```
final Object lock = new Object();
Thread t = new Thread(() -> {
    synchronized (lock) { lock.wait(); } });
t.start();
synchronized (lock) { lock.notifyAll(); }
t.join();
```

# Übung Dining Philosophers

Algorithmen implementieren und Eigenschaften (Liveness, Fairness) diskutieren/beweisen:

- ▶ Philosoph 0 ist Linkshänder
- ▶ ein Kellner (Platzanweiser), der immer nur maximal 4 Leute an den Tisch läßt

Realisierung des Modells

- ▶ in Java, in Promela, mit Petrinetz

# Übung Binärer Semaphor

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

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

Simulation allgemeiner Semaphor durch binären Semaphor

http:

[//www.csc.uvic.ca/~mcheng/460/notes/gensem.pdf](http://www.csc.uvic.ca/~mcheng/460/notes/gensem.pdf)

- ▶ Warum ist Solution 1 falsch,
- ▶ worin besteht Unterschied zu Solution 2?

Dazu Spin benutzen! Modelliere diesen Testfall: Semaphor beginnt bei 0. Jede Folge von  $N$  mal `signal` und  $N$  mal `wait` sollte erfolgreich sein (nicht blockieren).

```
int c = 0; int mutex = 1; int delay = 0; int done =
#define N 2
active[N] proctype Signaler() {
    atomic { mutex > 0; mutex=0 } c++;
    ... mutex = 1; }
active[N] proctype Waiter () {
    ... done++; }
ltl { eventually (done == N) }
```

# 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/people/simonpj/papers/stm/>

## 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 purposes)* USENIX 1996, <https://web.stanford.edu/~ouster/cgi-bin/papers/threads.pdf>



# Speicher-Transaktionen (Benutzung)

```
from <- atomically $ newTVar 10
atomically $ do x <- readTVar from
                if x < a then retry
                else writeTVar from (x-a)
```

- ▶ Transaktions-Variablen
- ▶ Lese- und Schreibzugriffe nur innerhalb einer Transaktion  
Zugriff außerhalb ist
  - ▶ statischer (Typ-)Fehler in Haskell
  - ▶ Laufzeitfehler in Clojure
- ▶ Transaktion wird atomar und isoliert ausgeführt
  - ▶ atomar: findet komplett statt oder überhaupt nicht
  - ▶ isoliert: ...<http://blog.franslundberg.com/2013/12/acid-does-not-make-sense.html>

# Speicher-Transaktionen in Promela/Spin

```
byte from = 10;
```

```
atomic { from >= a ; from = from - a }
```

# Speicher-Transaktionen (Implementierung)

- ▶ während der Transaktion:  
Zugriffe (Schreiben und Lesen) in Log schreiben
- ▶ am Ende (commit): prüfen, ob Log konsistent mit aktuellem Speicherzustand ist  
konsistent := die während der Transaktion gelesenen Werte stimmen mit den aktuellen überein
- ▶ ... wenn ja, dann schreiben (atomar)
- ▶ ..., wenn nicht, dann Transaktion wiederholen
- ▶ ... bei Wert-Änderung einer der gelesenen Variablen

Einzelheiten, Erweiterungen: <https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/STM>

# Deklaration von Nebenwirkungen in Typen

in Java (u.v.a.m.) ist der Typ nur ein Teil der Wahrheit:

```
public static int f (int x) {  
    y++ ; new File ("/etc/passwd").delete();  
    return x+1;  
}
```

in Haskell: Typ zeigt mögliche Nebenwirkungen an.  
damit kann man trennen:

- ▶ Aktion (IO Int)  
von Resultat (Int)
- ▶ Aktion, die in Außenwelt sichtbar ist (IO Int)  
von Aktion, die in Transaktion erlaubt ist (STM Int)

# 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/stm/2.2.0.1/doc/html/Control-Monad-STM.html>

# 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 Bull. 26, 1994.

Lösung mit STM in Peyton Jones: *Beautiful Concurrency*, 2007

# Philosophen mit STM

```
forM [ 1 .. num ] $ \ p -> forkIO $ forever $ do
  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
```

**kein Deadlock (trivial).** — nicht fair, siehe <http://thread.gmane.org/gmane.comp.lang.haskell.parallel/305>

**Quelltexte:** [https:](https://gitlab.imn.htwk-leipzig.de/waldmann/skpp-ws15)

[//gitlab.imn.htwk-leipzig.de/waldmann/skpp-ws15](https://gitlab.imn.htwk-leipzig.de/waldmann/skpp-ws15)

# Übung STM

- ▶ ein Haskell-Hauptprogramm schreiben,

```
main :: IO ()
```

```
main = do putStrLn "hello world"
```

kompilieren, ausführen (Optionen sind hier nicht nötig, aber später)

```
ghc -threaded -rtsospts -O2 Main.hs
```

```
./Main +RTS -N
```

- ▶ dining philosophers in Haskell, in <https://gitlab.imn.htwk-leipzig.de/waldmann/skpp-ws16>, kompilieren, ausführen. Diskutiere

```
https://mail.haskell.org/pipermail/
```

```
haskell-cafe/2015-November/122233.html Was passiert bei if p > 1 then ...?
```

- ▶ Nach Aktivieren der Anweisung

```
hSetBuffering stdout NoBuffering
```

wird deutlich, daß der gemeinsame Zugriff der

Philosophen auf `stdout` (in `putStrLn`) ebenfalls eine Konfliktquelle ist.

# STM in Clojure (Beispiele)

Clojure = LISP für JVM

```
(def foo (ref "bar"))  -- newTVar
```

```
(deref foo)           -- readTVar  
@foo
```

```
(ref-set foo "oof")   -- writeTVar  
(dosync (ref-set foo "oof"))
```

Quellen:

- ▶ Kap. 6 *Concurrency* aus: Stuart Halloway, *Programming Clojure*, Pragmatic Bookshelf, 2009;
- ▶ <http://clojure.org/refs>

# STM in Clojure (Sicherheit)

Transaktionsvariablen ohne Transaktion benutzen:

- ▶ Haskell: statischer Typfehler
- ▶ Clojure: Laufzeitfehler

IO innerhalb einer Transaktion:

- ▶ Haskell: statischer Typfehler
- ▶ Clojure: “I/O and other activities with side-effects should be avoided in transaction. . .”

Übung: ein Programm konstruieren, bei dem eine IO-Aktion innerhalb einer Transaktion stattfindet, aber die Transaktion nicht erfolgreich ist.

# Transaktion mit Nebenwirkung

## Transaktionen:

```
(def base 100)
(def source (ref (* base base)))
(def target (ref 0))
(defn move [foo]
  (dotimes [x base]
    (dosync (ref-set source (- @source 1))
            (ref-set target (+ @target 1))) ))
(def movers (for [x (range 1 base)] (agent nil)))
(dorun (map #(send-off % move) movers))
```

## Nebenwirkung einbauen:

```
(def c (atom 0)) ... (swap! c inc) ...
(sprintf c)
```

# STM und persistente Datenstrukturen

“The Clojure MVCC STM is designed to work with the persistent collections, and it is strongly recommended that you use the Clojure collections as the values of your Refs. Since all work done in an STM transaction is speculative, it is imperative that there be a low cost to making copies and modifications.”

“The values placed in Refs must be, or be considered, immutable!!”

Beispiel Suchbäume:

- ▶ destruktiv: Kind-Zeiger der Knoten verbiegen,
- ▶ persistent: neue Knoten anlegen.

Bsp: persistenter Suchbaum in Haskell



# 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 zusammengesetzte 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 -XX:+Pri
```

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

# 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 ()
}
```

# Spezifikation f. Concurrent Stacks

Stack-spezifisch:

- ▶ correct set semantics

allgemein:

- ▶ linearizability
- ▶ lock-free (lebendig), wait-free (fair)

vgl. Hendler, Shavit, Yerushalmi: *A Scalable Lock-free Stack Algorithm* (Sect. 5) (16th ACM Symp. on Parallelism in Algorithms and Architectures) <http://www.cs.bgu.ac.il/~hendlerd/papers/scalable-stack.pdf>

# Abstraktion, Linearisierbarkeit

- ▶ nebenläufige Implementierung  $N$  einer Datenstrukturspezifikation  $P$
- ▶ mit *abstraction map*  $a$  von  $N$  zu einer sequentiellen Implementierung  $S$
- ▶  $N$  heißt *linearisierbar*, falls es für jede nebenläufige Ausführung von  $N$  eine Folge von *Linearisierungspunkten*  $n_1, n_2, \dots$  gibt, so daß  $a(n_1), a(n_2), \dots$  eine  $P$ -korrekte Ausführung von  $S$  ist.

vgl. Shavit: *Art of Multiproc. Prog.* Sect. 9.3 *Concurrent Reasoning*



# 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/synchronization/pseudocode/queues.html`

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`

# Non-Blocking Übung

- ▶ CAS-Maschinenbefehler in JVM-Assemblercode
- ▶ Messungen für Zähler mit CAS, mit `synchronized`
- ▶ Testfälle für nebenläufige Stacks/Queues
- ▶ push/pop für non-blocking Stack
- ▶ enqueue/dequeue für non-blocking Queue

# Motivation

Motivation: zentrale Ausgabe von Tickets (mit eindeutigen und aufsteigenden Nummern).

mit höherem Durchsatz als mit einem *zentralen* Zähler

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

James Aspnes, Maurice Herlihy, and Nir Shavit.

*Counting networks*, JACM 41(5):1020–1048, Sept. 1994

<http://www.cs.yale.edu/homes/aspnes/papers/ahs-abstract.html>

wesentlicher Baustein: `AtomicBoolean.negate()`

# Spezifikation für Zählnetze

korrekte Behandlung der Token:

- ▶ Netzwerk mit  $n$  Eingängen,  $n$  Ausgängen, Tiefe  $d$
- ▶ jedes Token, das einen Eingang betritt, verläßt das Netzwerk nach  $\leq d$  Schritten an einem Ausgang (das Netzwerk vergißt und erfindet keine Token)

gute Verteilung der Token:

- ▶ (informal) bei *beliebiger* Verteilung der Token auf die Eingänge: jeder Ausgang wird (etwa) *gleich oft* benutzt.
- ▶ (formal) betrachte Anzahlen  $[x_1, \dots, x_n]$  der Token je Eingang, Anzahlen  $[y_1, \dots, y_n]$  der Token je Ausgang; im *Ruhezustand* ( $\sum_{i=1}^n x_i = \sum_{i=1}^n y_i$ ) soll gelten:  
 $[y_1, \dots, y_n]$  ist *Schrittfolge*:  $y_1 \geq \dots \geq y_n \geq y_1 - 1$

# Folgerung aus Spezifikation für Zählnetze

Satz: für jedes  $n > 0$ ,  $S \geq 0$  gibt es *genau eine* Schrittfolge  $[z_1, \dots, z_n]$  mit  $S = \sum z_i$ .

Satz: für *jeden* Zustand jedes Zählnetzes gilt:

- ▶ wenn  $\sum x_i - \sum y_i = D > 0$   
(es befinden sich noch  $D$  Token im Netz),
- ▶ dann gilt  $\forall i : z_i - \square \leq y_i \leq z_i$   
wobei  $[z_1, \dots]$  die (eindeutige) Schrittfolge mit  $\sum z_i = \sum x_i$

Folgerung: auch wenn der Ruhezustand nie eintritt, sind die Ausgänge gut verteilt

(hoher Durchsatz  $\implies$  kleines  $D \implies$  gute Verteilung)

# Netzwerke aus Verteilern

## Verteiler:

- ▶ 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.

## Eigenschaften/Fragen:

- ▶ jeder Verteiler ist ein Zählnetz für 2 Eingänge
- ▶ gibt es Zählnetze aus Verteilern (z. B. für 4 Eingänge)?
- ▶ kann man diese systematisch konstruieren?

# Bitonisches Zählen und Zusammenfügen (I)

Ansatz für Konstruktion eines  $2^k$ -Zählnetzes aus Verteilern:

- ▶ Zählnetze  $C$  benutzen Teilnetzwerke  $M$ , deren *Eingangsfolgen* (nach Induktion) Schrittfolgen sind.  
(vergleiche `mergesort`: die Funktion `merge` wird nur auf geordnete Folgen angewendet)
- ▶ Konstruktion der Zählnetze: Induktionsanfang:  $C_1(x_1) =$   
Induktionsschritt:  $C_{2n}(x_1, \dots, x_{2n}) =$   
 $C_n(x_1, \dots, x_n); C_n(x_{n+1}, \dots, x_{2n}); M_{2n}(x_1, \dots, x_n; x_{n+1}, \dots, x_{2n})$
- ▶ Konstruktion der Merge-Netze: (Spezifikation?)  
Induktionsanfang:  $M_2(x_1, x_2)$ ; Induktionsschritt?



## Bitonisches Zählen und Zusammenfügen (II)

Induktionsschritt:

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

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

Satz: jedes solche  $M_n$  erfüllt die Spezifikation.

Übung: konstruiere  $C_4, M_4$

Übung: Beweis für  $M_8$  mit Eingangsfolge

$(3, 3, 2, 2; 14, 13, 13, 13)$ , 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.

# Implementierung für Verteiler und Netze

## Plan:

```
struct Balancer {
    AtomicBoolean state;
    Balancer [Boolean] next;
}
traverse (Balancer b) {
    while (nicht fertig) {
        boolean i = b.state.getAndNegate();
        traverse(b.next[i]);    } }
```

## Aufgaben:

- ▶ implementiere `negate`
- ▶ implementiere Verteiler mit STM

# Anwendungen von Zählnetzen

<http://www.cs.yale.edu/homes/aspnes/papers/ahs-abstract.html> **Section 5**

- ▶ **verteiltes Zählen**  
 $n$  Ein/Ausgänge, an jedem Ausgang ein Zähler mit Schrittweite  $n$
- ▶ **verteilter Producer/Consumer-Puffer**  
benutzt zwei Netze der Breite  $n$  zum verteilten Zählen sowie  $n$  1-Element-Container
- ▶ **Synchronisationsbarriere** (vgl. CyclicBarrier)

# Übung Zählnetze

Beweise: die folgenden Bedingungen sind äquivalent:

- ▶  $(x_1, \dots, x_n)$  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  $\sum \text{odd}(x)$ ,  $\sum(x)/2$ ,  $\sum \text{even}(x)$ ?  
(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  $i$ ?

periodische Zählnetze

# 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/thoare/>
- ▶ 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:

- ▶  $\text{STOP} \in \mathbb{P}$ ,
- ▶ wenn  $e \in E$  und  $P \in \mathbb{P}$ , dann  $(e \rightarrow P) \in \mathbb{P}$
- ▶ wenn  $P_1, P_2 \in \mathbb{P}$ , dann sind in  $\mathbb{P}$ :
  - ▶ Nacheinanderausführung:  $P_1; P_2$
  - ▶ Auswahl, intern:  $P_1 \sqcap P_2$ , extern:  $P_1 \square P_2$
  - ▶ nebenläufige Ausführung  
mit Kommunikationsalphabet  $C \subseteq E$ :  
 $P_1 \parallel_C P_2$
- ▶ Wiederholung:  $P_1^* \in \mathbb{P}$

# CSP: Semantik

Semantik eines Prozesses  $P \in \mathbb{P}(E)$  definiert durch:

- ▶ Definition des Zustandsübergangs-Systems  $A(P)$ 
  - ▶ endlicher Automat mit Alphabet  $E$  und  $\epsilon$ -Übergängen
  - ▶ Zustände sind Prozesse (d.h. Terme)
  - ▶  $P$  ist initial, alle Zustände sind final (akzeptierend)
  - ▶ Übergänge beschrieben durch Term-Ersetzungs-Regeln
- ▶ Definition der Semantik von  $A(P)$ , zwei Möglichkeiten:
  - ▶ die Spur-Semantik (= die Sprache von  $A(P)$ )
  - ▶ die Ablehnungs-Semantik



# CSP: von Prozess zu Automat

Übergangsrelation von  $A(P)$  definiert durch Regeln zu

▶ Nacheinanderausführung:

▶  $(a \rightarrow P) \xrightarrow{a} P$

▶  $(\text{STOP}; Q) \xrightarrow{\epsilon} Q,$

▶ wenn  $P \xrightarrow{a} P'$ , dann  $(P; Q) \xrightarrow{a} (P'; Q),$

(vgl. Nil, Cons, Append für Listen)

▶ Wiederholung:

▶  $P^* \xrightarrow{\epsilon} \text{STOP}, \quad P^* \xrightarrow{\epsilon} (P; P^*).$

(vgl. Kleene-Hülle als Sprachoperation)

▶ sowie (nächste Folien) Kommunikation, Verzweigung

# Regeln zur Kommunikation

das Ereignis gehört zum Kommunikations-Alphabet:  
beide Prozesse führen es gemeinsam (synchron) aus

$$\blacktriangleright a \in C \wedge P \xrightarrow{a} P' \wedge Q \xrightarrow{a} Q' \Rightarrow (P \parallel_C Q) \xrightarrow{a} (P' \parallel_C Q'),$$

das Ereignis gehört nicht zum Kommunikations-Alphabet oder  
ist ein  $\epsilon$ -Übergang: einer der beiden Prozesse führt es aus (der  
andere wartet)

$$\blacktriangleright (a = \epsilon \vee a \in E \setminus C) \wedge P \xrightarrow{a} P' \Rightarrow (P \parallel_C Q) \xrightarrow{a} (P' \parallel_C Q),$$

$$\blacktriangleright (a = \epsilon \vee a \in E \setminus C) \wedge Q \xrightarrow{a} Q' \Rightarrow (P \parallel_C Q) \xrightarrow{a} (P \parallel_C Q'),$$

definiert *synchrone* Kommunikation, realisiert u.a. in  
Ada (Rendezvous), Scala (Operation !?),  
Go (Kanal mit Kapazität 0).

# Regeln für Auswahloperatoren

- ▶ interne Auswahl (Nichtdeterminismus)

$$P \sqcap Q \xrightarrow{\epsilon} P, \quad P \sqcap Q \xrightarrow{\epsilon} Q.$$

- ▶ externe Auswahl

$$P \xrightarrow{a} P' \Rightarrow (P \square Q) \xrightarrow{a} P', \quad Q \xrightarrow{a} Q' \Rightarrow (P \square Q) \xrightarrow{a} Q'.$$

Beispiel: (mit verkürzter Notation  $a$  für  $a \rightarrow \text{STOP}$ )

- ▶  $P_1 = (a \sqcap b)$ :  $A(P_1) = \text{STOP} \xleftarrow{b} b \xleftarrow{\epsilon} P_1 \xrightarrow{\epsilon} a \xrightarrow{a} \text{STOP}$
- ▶  $P_2 = (a \square b)$ :  $A(P_2) = \text{STOP} \xleftarrow{b} P_2 \xrightarrow{a} \text{STOP}$

diese Automaten sind verschieden, aber die Sprachen stimmen überein.

# Verschiedene Prozeß-Semantiken

- ▶ Spur-Semantik:

Menge der (unvollständigen) Spuren (*partial traces*)  
(jeder Automatenzustand ist akzeptierend)  
gestattet keine Beschreibung von Verklemmungen  
(*deadlocks*), keine Unterscheidung von interner und  
externer Auswahl, deswegen

- ▶ Ablehnungs-Semantik

zur genaueren Beschreibung des Prozeßverhaltens

Bemerkung: wenn man das nicht klar definiert,  
dann beginnt das große Rätselraten darüber,  
was *Nichtdeterminismus* für Prozesse bedeuten soll,  
vgl. <http://lambda-the-ultimate.org/node/4689>

# Ablehnungs-Semantik

Ab-Semantik eines Prozesses ist Menge von Paaren von

- ▶ partieller Spur  $s \in E^*$
- ▶ und Folge-Menge  $F \subseteq E$  (mögliche nächste Ereignisse)

$(s, F) \in \text{Ab}(P) : \iff \exists Q : P \xrightarrow{s} Q \wedge F = \{e \mid \exists R : Q \xrightarrow{e} R\}$ .

Beispiel: Ab-Semantik ist genauer als Sp-Semantik:

- ▶  $\text{Sem}_{\text{Sp}}(b \square c) = \text{Sem}_{\text{Sp}}(b \sqcap c) = \{\epsilon, b, c\}$
- ▶  $\text{Sem}_{\text{Ab}}(b \square c) = \{(\epsilon, \{b, c\}), (b, \emptyset), (c, \emptyset)\}$
- ▶  $\text{Sem}_{\text{Ab}}(b \sqcap c) = \{(\epsilon, \{b, c\}), (\epsilon, \{b\}), (b, \emptyset), (\epsilon, \{c\}), (c, \emptyset)\}$

# 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/Tasking](http://en.wikibooks.org/wiki/Ada_Programming/Tasking)  
[http://www.adaic.org/resources/add\\_content/standards/05aarm/html/AA-9-7-1.html](http://www.adaic.org/resources/add_content/standards/05aarm/html/AA-9-7-1.html)



# Plan

betrachten Zustandsübergangssysteme allgemein (Beispiele: endliche Automaten, Petri-Netze, CSP)

Semantiken und durch sie definierte Äquivalenzen:

- ▶ Spur-Semantik  $(S) \subseteq \Sigma^*$   
 $S_1$  spur-äquivalent zu  $S_2$ , falls  $\text{Sp}(S_1) = \text{Sp}(S_2)$ .
- ▶ Ablehnungs-Semantik  $\text{Ab}(S) \subseteq \Sigma^* \times \text{Pow}(\Sigma)$   
 $S_1$  ablehnungs-äquivalent zu  $S_2$ , falls  $\text{Ab}(S_1) = \text{Ab}(S_2)$ .
- ▶ Bisimulation:  $S_1$  *bisimilar* zu  $S_2$ , falls eine Relation (*Bisimulation*)  $R \subseteq \text{states}(S_1) \times \text{states}(S_2)$  mit bestimmten Eigenschaften existiert

# Definition

Zustandsübergangssystem  $S = (\Sigma, Q, T, i)$   
(Alphabet  $\Sigma$ , Zustandsmenge  $Q$ ,  
Transitionen  $T \subseteq Q \times \Sigma \times Q$ , Startzustand  $i \in Q$ )

Def:  $R \subseteq Q_1 \times Q_2$  ist *Bisimulation* zwischen  $S_1$  und  $S_2$ , falls:

- ▶ Vor- und Nachbereich groß genug:  
domain  $R = Q_1$ , range  $R = Q_2$
- ▶ Startzustände sind bisimilar:  $(i_1, i_2) \in R$
- ▶  $S_1$ -Transitionen durch  $S_2$ -Transitionen simuliert:  
 $\forall (p_1, p_2) \in R : \forall (p_1, a, q_1) \in T_1 :$   
 $\exists q_2 : (p_2, a, q_2) \in T_2 \wedge (q_1, q_2) \in R$
- ▶  $S_2$ -Transitionen durch  $S_1$ -Transitionen simuliert  
Ü: Diagramm zeichnen, Formel hinschreiben

# Beispiele, Kommentar

- ▶ Bisimulation kann Schleifen verschiedener Länge nicht voneinander unterscheiden, falls alle Schleifenknoten gleich aussehen (Beispiel)
- ▶ man kann in  $S$  alle Schleifen „ausrollen“ und erhält einen Baum  $T$ , der bisimilar zu  $S$  ist
- ▶  $T$  ist im allgemeinen unendlich, deswegen möchte man doch mit endlichem  $S$  rechnen.

# Bestimmung einer Bisimulation (Plan)

- ▶ Eingabe: Systeme  $(S_1, S_2)$
- ▶ berechne Folge von Relationen  $R_0, R_1 \dots \subseteq Q_1 \times Q_2$   
wobei  $(p_1, p_2) \in R_k \iff p_1$  in  $S_1$  und  $p_2$  in  $S_2$  verhalten sich für Transitionsfolgen der Länge  $\leq k$  „gleich“
- ▶ Folge ist monoton fallend bzgl. Inklusion:  
 $Q_1 \times Q_2 = R_0 \supseteq R_1 \supseteq R_2 \supseteq \dots$
- ▶ falls diese Folge schließlich stationär ist ( $\exists n : R_n = R_{n+1}$ ), dann teste, ob dieses  $R_n$  eine Bisimulation für  $(S_1, S_2)$  ist.

Sätze: Korrektheit, Vollständigkeit,  
Termination für endliche  $Q_1, Q_2$ .

vergleiche: Verfahren zur Minimierung von Automaten (Tabelle zur Markierung nicht äquivalenter Zustände)

# Bestimmung einer Bisimulation (Impl.)

aus Definition „ $R$  ist Bisimulation“:

- ▶  $S_1$ -Transitionen durch  $S_2$ -Transitionen simuliert:

$$\forall (p_1, p_2) \in R : \forall (p_1, a, q_1) \in T_1 :$$

$$\exists q_2 : (p_2, a, q_2) \in T_2 \wedge (q_1, q_2) \in R$$

- ▶ und symmetrisch ( $1 \leftrightarrow 2$ )

leite Verfeinerungsverfahren ab:

gegeben  $R_k$ , definiere  $R_{k+1}$  durch:

$(p_1, p_2) \in R_{k+1}$ , falls  $(p_1, p_2) \in R_k$  und

- ▶  $\forall (p_1, a, q_1) \in T_1 : \exists q_2 : (p_2, a, q_2) \in T_2 \wedge (q_1, q_2) \in R_k$
- ▶ und symmetrische Bedingung (tausche  $1 \leftrightarrow 2$ )

# Bisimulation (Übung)

- ▶ autotool-Aufgabe
- ▶ zwei Prozesse, die ablehnungs-äquivalent sind, aber nicht bisimilar

# Kommunikations-Kanäle

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

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

Bsp. in Go: (<http://golang.org>)

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

# Kanäle in Haskell

Kanal ist *typisiert*, FIFO, *unbeschränkt*.

```
data Chan a -- abstrakt
newChan    :: IO (Chan a)
writeChan ::
readChan  ::
```

Dok.: <http://www.haskell.org/ghc/docs/latest/html/libraries/base/Control-Concurrent-Chan.html>

## Übungen

- ▶ Implementierung ansehen
- ▶ Anwendung: Aufsammeln von Teilergebnissen
- ▶ Anwendung: Mergesort in Aktor-Style
- ▶ vergleiche mit `Control.Concurrent.STM.TChan`



# Haskell: MVar

ist Kanal der Kapazität 1

```
data MVar a = ...
```

```
takeMVar :: MVar a      -> IO a  
-- blockiert, wenn leer
```

```
putMVar  :: MVar a -> a -> IO ()  
-- blockiert, wenn voll
```

# Actors (Scala)

- ▶ Briefkasten ist nicht typisiert
- ▶ Nachrichten sind typisiert

<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 Akteur soll nicht blockieren  
... sondern lieber Arbeit an andere Akteure weitergeben
- ▶ kommuniziere mit Akteuren nur durch Nachrichten  
... und nicht durch gemeinsame Variablen
- ▶ Nachrichten sollten *immutable* sein  
... sonst Gefahr von inkonsistenten Daten
- ▶ Nachrichten sollten *self-contained* sein  
... damit der Akteur 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`

# Kanäle in Spin

- ▶ Syntax (!, ?), Semantik (Kapazität, Blockieren)
- ▶ Beispiel: Simulation eines Semaphors durch einen Kanal:

```
chan s = [1] of { byte }; byte c = 0;
active [3] proctype P () { byte x;
    do :: s ! 0 ; c++ ; c-- ; s ? x ;
    od
}
```

- ▶ Verifikation des gegenseitigen Ausschlusses (bei beliebig vielen Prozessen)  $1t1$  { [] (c < 2) }  
der Fairness (bei zwei Prozessen)?

# Übung: Kanäle in Go

Sprachdefinition: <http://golang.org/>  
Compiler/Runtime:

- ▶ **google:** `go run hello.go`
- ▶ **gcc:** `gcc-go -o hello hello.go ; ./hello`

Kanäle:

- ▶ Syntax (Deklaration, Benutzung)
- ▶ Kapazität
- ▶ Schließen von Kanälen

Übung:

- ▶ berechne Summe der Bitcounts von 0 bis  $2^n - 1$
- ▶ verteile Rechnung auf  $p$  Prozesse

# Futures in Java

`submit` startet eine asynchrone Berechnung,  
`get` ist der blockierende Zugriff auf das Resultat.  
Implementierung könnte einen Kanal benutzen  
(die Rechnung schreibt, `get` liest)

```
package java.util.concurrent;
interface Callable<R> { R call() }
interface ExecutorService {
    <R> Future<R> submit (Callable<R>)
    void shutdown() }
interface Future<R> { R get() }
class Executors {
    ExecutorService newFixedThreadPool(int) }
```

Ü: Bitcount-Summation. `Stream<Future<Integer>>`

# Verteiltes Rechnen

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

Realisierungen:

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



# Erlang

*Ericsson Language*, <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

- ▶ keine Sprache, sondern Bibliothek  
(= eDSL, *eingebettete* domainspezifische Sprache)
- ▶ Semantik angelehnt an Erlang-Prozesse
- ▶ anwendbar, wenn alle Knoten binärkompatibel sind und identische Maschinenprogramme ausführen (dann können Zeiger in diese Programme in Nachrichten transportiert werden)

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/papers/parallel/>

<http://haskell-distributed.github.io/>

# 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
```

# Beispiele/Übung

- ▶ Quelltexte VL KW 50

`https://gitlab.imn.htwk-leipzig.de/waldmann/skpp-ws16/tree/master/kw50`

- ▶ Ergänzen Sie den Quelltext des verteilten Mergesort (ms.go)

Nur zum Zweck der Übung des Nachrichtentransport über Kanäle, für paralleles Sortieren gibt es besser geeignete Verfahren.

# 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)
- ▶ Haskell benutzt Bedarfsauswertung.  
Strategie-Kombinatoren und -Annotationen erzwingen frühere/verteilte Auswertung von Teilausdrücken:
  - ▶ Kombinatoren: `par X (pseq Y (f X Y))`
  - ▶ Strategie-Annotationen: `xs `using` parList rseq`

# 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  $\Rightarrow$  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 (struct) wird Programm (thunk) gespeichert, das  
den Wert ausrechnen kann  
(ähnlich einer `Future`, die noch nicht gestartet wurde)
- ▶ Konstruktor wird erst bestimmt,  
wenn er für Pattern Matching benötigt wird  
(Vergleich mit einem Konstruktor eines Musters in `case`)
- ▶ dann wird thunk durch Wert überschrieben
- ▶ Wert kann selbst wieder thunks enthalten



# Normalformen

- ▶ ein Objekt, das kein thunk ist, heißt (schwache) *Kopfnormalform* (whnf)  
d.h., der *oberste* Konstruktor ist bestimmt  
Teilobjekte können thunks sein
- ▶ ein Objekt, das kein thunk ist und keine thunks enthält,  
heißt *Normalform*  
d.h., *alle* Konstruktoren sind bestimmt
- ▶ Zahlen, Zeichen sind auch Normalformen  
aber `Typ Int` bedeutet „thunk oder Zahl“:  
`data Int = GHC.Types.I# GHC.Prim.Int#`

# Experiment zu Liste, Sequence, whnf

```
ghci
:set +s

let xs = [1..10^6] :: [Int]
seq xs () ; head xs ; last xs ; last xs

import qualified Data.Sequence as S
let ys = S.fromList xs
seq ys () ; S.index ys 0
S.index ys (10^6-1)
```

# Bedarfsauswertung und Parallelisierung

- ▶ GHC RTS verwaltet *spark pool*,  
Spark ist (Zeiger auf) thunk, Pool ist Menge von Sparks
- ▶ parallele worker threads bestimmen whnf der sparks
- ▶ Spark (für  $x$ ) wird erzeugt durch `par x y`
- ▶ Schicksal der Sparks (Statistik: `./Main +RTS -N -s`)
  - ▶ *converted*: worker überschreibt thunk durch whnf  
(das ist anzustreben: worker hilft Hauptprogramm)
  - ▶ *fizzled*: Hauptprogramm überschreibt thunk durch whnf  
(das ist schlecht: worker hat umsonst gearbeitet)
  - ▶ *garbage collected*: whnf wurde nicht benötigt

# Beispiel: Mergesort

Sequentieller Algorithmus, wesentlicher Teil:

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

parallelisiert durch: `import Control.Parallel`

```
.. in par sh $ pseq st $ merge sh st
```

- ▶ Datentyp `[a]` nützt hier wenig:  
Cons ist lazy, `whnf` enthält ggf. nur ein Cons.
- ▶ `Data.Sequence.Seq a` ist besser  
ist balancierter Baum, deswegen `whnf ≈ nf`

## 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 durch Strategie-Annotation

```
withStrategy ( parListChunk 100000 rdeepseq )
  ( map prime [lo..hi] )
```

getrennte Beschreibung von *Wert* und *Strategie*  
Beschreibungssprache (EDSL) für Strategien

<http://hackage.haskell.org/package/parallel/docs/Control-Parallel-Strategies.html>

# Parallel LINQ

## Beispiel:

```
(from n in Enumerable.Range(10, hi-10)
                          .AsParallel()
 where Prime(n)  select true).Count ();
```

## Typen:

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

<http://msdn.microsoft.com/en-us/library/dd997425.aspx>

## Übung:

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

# Übung

- ▶ Laufzeitanalyse mit Threadscope

```
ghc -O2 -eventlog -threaded Pr.hs  
./Pr 6 +RTS -N2 -l  
threadscope Pr.eventlog
```

- ▶ Primzahlen:

- ▶ finde beste *chunk size* für `map isprime [..]`
- ▶ warum ist parallele Auswertung innerhalb von `isprime` unzweckmäßig?

- ▶ bitcounts summieren

# Motivation

*Beispiel* Summe eine Liste durch beliebige Klammerung von zweistelligen Additionen

$$\text{sum } [3, 1, 2, 4] = ((3+1)+2)+4 = 3+(1+(2+4))$$

wähle den Berechnungsbaum, der am besten zu verfügbarer Hardware (Anzahl Prozessoren) paßt

$$\dots = (3+1)+(2+4)$$

*Verallgemeinerung* statt Addition: beliebige assoziative Operation, nicht notwendig kommutativ, d.h. Blatt-Reihenfolge muß erhalten bleiben



# Monoide

- ▶ eine Struktur  $A$  mit Operation  $\circ_A$  und Element  $1_A$  heißt *Monoid*, falls:
  - ▶  $\circ_A$  ist assoziativ
  - ▶  $1_A$  ist links und rechts neutral für  $\circ_A$
- ▶ Beispiele:  
Listen, Verkettung, leerer Liste  
natürliche Zahlen, Addition, 0
- ▶ Monoid-Morphismen von Listen lassen sich flexibel parallelisieren
- ▶ Beispiele: Länge, Summe.

# Homomorphismen

homo-morph = gleich-förmig

Signatur  $\Sigma$  (= Menge von Funktionssymbolen)

Abbildung  $h$  von  $\Sigma$ -Struktur  $A$  nach  $\Sigma$ -Struktur  $B$  ist

Homomorphismus, wenn:

$\forall f \in \Sigma, x_1, \dots, x_k \in A :$

$$h(f_A(x_1, \dots, x_k)) = f_B(h(x_1), \dots, h(x_k))$$

Beispiel:

- ▶  $\Sigma =$  Monoid-Signatur (Element 1, binäre Operation  $\circ$ )
- ▶  $A =$  List a (Listen) mit  $1_A = \text{Nil}$ ,  $\circ_A = \text{append}$
- ▶  $B = \mathbb{N}$  (Zahlen) mit  $1_B = 0$ ,  $\circ_B = \text{plus}$
- ▶  $h = \text{length}$

# Homomorphismen von Listen

Abbildung  $[a] \rightarrow b$  ist gegeben durch

- ▶ Bild der leeren Liste  $:: b$
- ▶ Bild der Einerliste  $:: a \rightarrow b$
- ▶ Verknüpfung  $:: b \rightarrow b \rightarrow b$

```
foldb :: b -> (a->b)->(b->b->b) -> [a]-> b
foldb e g f xs = case xs of
  [] -> e ; [x] -> g x
  _ -> let (l,r) = splitAt ... xs
        in f (foldb e g f l) (foldb e f g r)
```

**Satz:**  $f$  assoziativ und  $e$  links und rechts neutral für  $f \Rightarrow$   
 $\text{foldb } e \text{ } g \text{ } f$  ist Monoid-Homomorphismus von  
 $([a], [], ++)$  nach  $(b, e, f)$

# Maximale Präfix-Summe

```
mps :: [Int] -> Int
```

```
mps xs = maximum $ map sum $ inits xs
```

```
mps [1,-1,0,2,0,-1] = maximum [0,1,0,0,2,2,1] = 2
```

ist kein Homomorphismus (Gegenbeispiel?) aber das:

```
mpss :: [ Int ] -> ( Int, Int )
```

```
mpss xs = ( mps xs, sum xs )
```

**Bestimme die Verknüpfung in  $(\text{Int}, \text{Int})$**

**(die Darstellung als `foldb`)**

**beweise Assoziativität**

# Sequentielle Folds

für sequentielle Berechnung sind geeignet

```
foldr :: ( a -> b -> b) -> b -> [a] -> b
foldr f e xs = case xs of
  [] -> e ;  x : ys -> f x (foldr f e ys)
```

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f e xs = case xs of
  [] -> e ;  x : ys -> foldl f (f e x) ys
```

`foldl` paßt zum Iterator-Muster (Berechnung über einen Stream, mit einem Akkumulator), vgl. `Aggregate` in C#/Linq

# Homomorphie-Sätze

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/download?doi=10.1.1.45.2247&rep=rep1&type=pdf
- ▶ Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, Masato Takeichi: *Automatic Inversion Generates Divide-and-Conquer Parallel Programs*, PLDI 2007.

## Übung Assoc.

- ▶ ist  $f :: [a] \rightarrow \text{Bool}$  mit  $f \text{ xs} =$  „die Länge von  $\text{xs}$  ist gerade“ ein Morphismus?  
desgl. für „... ist durch 3 teilbar“  
wenn nein: Gegenbeispiel lt. Definition,  
wenn ja: Implementierung mit `foldb`

- ▶ desgl. für

$f :: [a] \rightarrow \text{Maybe } a$

$f \text{ xs} = \text{case } \text{xs} \text{ of}$

$[\ ] \rightarrow \text{Nothing} ; \text{ x:ys} \rightarrow \text{Just } x$

- ▶  $\text{mss} :: [\text{Int}] \rightarrow \text{Int}$  maximale Segmentsumme  
(über alle zusammenhängenden Teilfolgen)  
durch ein assoziative Verknüpfung auf angereicherter  
Struktur (Hinweis: 3-Tupel von Zahlen)
- ▶  $\text{carry} :: [(\text{Bool}, \text{Bool})] \rightarrow \text{Bool}$  Übertrag bei  
Addition von Binärzahlen  
Hinweis: betrachte

$f :: [(\text{Bool}, \text{Bool})] \rightarrow (\text{Bool}, \text{Bool})$  mit

Spezifikation



# Diskussion: Kommutativität

Methode aus PLINQ (<http://msdn.microsoft.com/en-us/library/ff963547.aspx>)

```
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/2008/01/22/7211660.aspx>) — warum?

# 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.  
<https://research.google.com/archive/mapreduce.html>
- ▶ Ralf Lämmel: *Google's MapReduce programming model - Revisited*, Science of Computer Programming - SCP , vol. 70, no. 1, pp. 1-30, 2008 <https://userpages.uni-koblenz.de/~laemmel/MapReduce/paper.pdf>

# 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)] ) -- ^ distribute
  -> ( 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, IntWritable>{
    public void map(Object key, Text value, Context
public static class IntSumReducer
    extends Reducer<Text,IntWritable,Text,IntWritable>{
    public void reduce(Text key, Iterable<IntWritable>
    }
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/
WordCount.java
```

# 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  $\text{link} :: (\text{URL}, \text{URL}) \rightarrow \text{Double}$  mit  $\text{link}(u, v) =$  Wahrscheinlichkeit, daß der Besucher von  $u$  zu  $v$  geht.
- ▶ Gesucht: Vektor  $w :: \text{URL} \rightarrow \text{Double}$  mit  $w * \text{link} = w$

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 besetzte) Matrizen mit map/red?
- ▶ Map/Reduce mit Hadoop:  
[http://www.michael-noll.com/tutorials/  
running-hadoop-on-ubuntu-linux-multi-node-cluster/](http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-multi-node-cluster/)

# Beispiel

- ▶ Aufgabe: gesucht ist
  - ▶ Konfiguration von  $P$  Punkten im Einheitsquadrat,
  - ▶ deren minimaler gegenseitiger Abstand maximal ist.

( <http://www2.stetson.edu/~efriedma/cirinsqu/> )
- ▶ Lösungsplan: zufällige lokale Suche
- ▶ Teilaufgabe dabei: bewerte eine Liste von  $K$  Konfigurationen
- ▶ Parallelisierungsmöglichkeiten:
  - ▶ Bewertung für alle Konfigurationen gleichzeitig
  - ▶ Abstandsberechnung für alle Punktepaare gleichzeitig
- ▶ das ist *massiv* parallel ( $K = 128, P = 16$ ) ...  
solche Prozessoren gibt es *preiswert*  
CUDA ist eine C-Erweiterung zur Programmierung dafür



# Beispiel

- ▶ Aufgabe: gesucht ist
  - ▶ Konfiguration von  $P$  Punkten im Einheitsquadrat,
  - ▶ deren minimaler gegenseitiger Abstand maximal ist.

( <http://www2.stetson.edu/~efriedma/cirinsqu/> )
- ▶ Lösungsplan: zufällige lokale Suche
- ▶ Teilaufgabe dabei: bewerte eine Liste von  $K$  Konfigurationen
- ▶ Parallelisierungsmöglichkeiten:
  - ▶ Bewertung für alle Konfigurationen gleichzeitig
  - ▶ Abstandsberechnung für alle Punktepaare gleichzeitig
- ▶ das ist *massiv* parallel ( $K = 128, P = 16$ ) ...  
solche Prozessoren gibt es *preiswert*  
CUDA ist eine C-Erweiterung zur Programmierung dafür

# Beispiel

- ▶ Aufgabe: gesucht ist
  - ▶ Konfiguration von  $P$  Punkten im Einheitsquadrat,
  - ▶ deren minimaler gegenseitiger Abstand maximal ist.
- ( <http://www2.stetson.edu/~efriedma/cirinsqu/> )
- ▶ Lösungsplan: zufällige lokale Suche
- ▶ Teilaufgabe dabei: bewerte eine Liste von  $K$  Konfigurationen
- ▶ Parallelisierungsmöglichkeiten:
  - ▶ Bewertung für alle Konfigurationen gleichzeitig
  - ▶ Abstandsberechnung für alle Punktepaare gleichzeitig
- ▶ das ist *massiv* parallel ( $K = 128, P = 16$ ) ...  
solche Prozessoren gibt es *preiswert*  
CUDA ist eine C-Erweiterung zur Programmierung dafür

## Sequentielle und parallele Lösung (Ansatz)

```
for (int k = 0; k < K; k++) {  
    float mindist = +infinity;  
    for (int p = 0; p < P; p++) {  
        for (int q = p + 1; q < P; q++ ) {  
            float s = abstand (c[k][p], c[k][q]);  
            mindist = MIN (mindist, s); } }  
    c[k].fitness = mindist; }  
}
```

```
dim3 blocks (K, 1) ; dim3 threads (P, P);  
kernel <<<blocks,threads>>> (c);  
__global__ kernel (config *c) {  
    int k = blockIdx.x;  
    int p = threadIdx.x; int q = threadIdx.y;  
    if (p < q) { float s = abstand (c[k][p], c[k][q]); }  
    // Berechnug des MIN?  
}
```

# Threadkommunikation

Threads in einem Block haben gemeinsamen *shared memory*

```
__global__ kernel (config *c) {  
    __shared__ float dist [P*P];  
  
    if (p < q) dist[p*P+q] = abstand(c[k][p], c[k][q]);  
    else      dist[p*P+q] = +infinity;
```

Synchronisationspunkt für *alle* (!) Threads in einem Block:

```
__syncthreads();
```

erst danach kann man Ergebnisse aufsammeln

# Threadkommunikation

Threads in einem Block haben gemeinsamen *shared memory*

```
__global__ kernel (config *c) {  
    __shared__ float dist [P*P];  
  
    if (p < q) dist[p*P+q] = abstand(c[k][p], c[k][q]);  
    else      dist[p*P+q] = +infinity;
```

Synchronisationspunkt für *alle* (!) Threads in einem Block:

```
__syncthreads();
```

erst danach kann man Ergebnisse aufsammeln

# Binäre Reduktion

sequentiell (linear)

```
float accu = +infty;  
for ( ... ) { accu = MIN (accu, dist[ ..]); }
```

parallel (logarithmisch)

```
int tid = P * p + q; int gap = threadsPerBlock / 2;  
while ( gap > 0 ) {  
    if (dist[tid] > dist[tid + gap])  
        dist[tid] = dist[tid + gap];  
    __syncthreads ();  
    gap /= 2;  
}
```

Ergebnis verarbeiten

```
if (0 == tid) c[k] = dist[0];
```

# Binäre Reduktion

sequentiell (linear)

```
float accu = +infty;  
for ( ... ) { accu = MIN (accu, dist[ ..]); }
```

parallel (logarithmisch)

```
int tid = P * p + q; int gap = threadsPerBlock / 2;  
while ( gap > 0 ) {  
    if (dist[tid] > dist[tid + gap])  
        dist[tid] = dist[tid + gap];  
    __syncthreads ();  
    gap /= 2;  
}
```

Resultat verarbeiten

```
if (0 == tid) c[k] = dist[0];
```

# Binäre Reduktion

sequentiell (linear)

```
float accu = +infty;  
for ( ... ) { accu = MIN (accu, dist[ ..]); }
```

parallel (logarithmisch)

```
int tid = P * p + q; int gap = threadsPerBlock / 2;  
while ( gap > 0 ) {  
    if (dist[tid] > dist[tid + gap])  
        dist[tid] = dist[tid + gap];  
    __syncthreads ();  
    gap /= 2;  
}
```

Resultat verarbeiten

```
if (0 == tid) c[k] = dist[0];
```



# Datentransport zwischen Host und Device

```
config * dev_c ;  
cudaMalloc ( (void**) &dev_c, K*sizeof(config) );  
  
cudaMemcpy ( dev_c, c, K*sizeof(config),  
            cudaMemcpyHostToDevice );  
  
...  
cudaMemcpy ( c, dev_c, K*sizeof(config),  
            cudaMemcpyDeviceToHost );  
cudaFree (dev_c);
```

... das ist teuer ( $\Rightarrow$  möglichst wenige Daten transportieren)

# Zusammenfassung CUDA

CUDA-Programmierung ist

- ▶ programmiersprachlich einfach
- ▶ algorithmisch anspruchsvoll  
(imperative parallele Programmierung)

jeweils projektspezifisch sind festzulegen

- ▶ was läuft auf Device, was auf Host?
- ▶ Abmessung der Threadblöcke auf dem Device?
- ▶ Host realisiert Software-API (Steuerprogramm in Hochsprache)

Alternativen:

- ▶ OpenCL <https://www.khronos.org/opencv/>
- ▶ accelerate/CUDA-backend  
<http://hackage.haskell.org/package/accelerate>

# Zusammenfassung CUDA

CUDA-Programmierung ist

- ▶ programmiersprachlich einfach
- ▶ algorithmisch anspruchsvoll  
(imperative parallele Programmierung)

jeweils projektspezifisch sind festzulegen

- ▶ was läuft auf Device, was auf Host?
- ▶ Abmessung der Threadblöcke auf dem Device?
- ▶ Host realisiert Software-API (Steuerprogramm in Hochsprache)

Alternativen:

- ▶ OpenCL <https://www.khronos.org/opencv/>
- ▶ accelerate/CUDA-backend  
<http://hackage.haskell.org/package/accelerate>

# Zusammenfassung CUDA

CUDA-Programmierung ist

- ▶ programmiersprachlich einfach
- ▶ algorithmisch anspruchsvoll  
(imperative parallele Programmierung)

jeweils projektspezifisch sind festzulegen

- ▶ was läuft auf Device, was auf Host?
- ▶ Abmessung der Threadblöcke auf dem Device?
- ▶ Host realisiert Software-API (Steuerprogramm in Hochsprache)

Alternativen:

- ▶ OpenCL <https://www.khronos.org/opencv/>
- ▶ accelerate/CUDA-backend  
<http://hackage.haskell.org/package/accelerate>

# Literatur CUDA

- ▶ David B Kirk, Wen-mei W Hwu, *Programming Massively Parallel Processor*, Morgan Kaufmann, 2010
- ▶ Jason Sanders, Edward Kandrot: *CUDA by Example*, Addison-Wesley, 2011 <http://developer.nvidia.com/cuda-example-introduction-general-purpose-gpu-programming>

# Parallele Algorithmen

- ▶ in SKPP haben wir programmiersprachliche Konzepte betrachtet, um parallele Algorithmen zu implementieren.
- ▶ woher kommen die Algorithmen?  
paralleles Suchen, Sortieren, Matrixmultiplizieren, ...

Bsp: `http:`

`//www.iti.fh-flensburg.de/lang/algorithmen/  
sortieren/twodim/shear/shearsorten.htm`

dazu braucht man eine eigenen Vorlesung, vgl.

- ▶ Joseph JaJa: *Introduction to Parallel Algorithms* (Addison-Wesley, 1992) ISBN-10: 0201548569
- ▶ *Algorithm Design: Parallel and Sequential* (CMU, aktuell)  
`http://www.parallel-algorithms-book.com/`

# Komplexitätstheorie

... für parallele Algorithmen

Klassen:

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

Reduktionen:

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

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

# Zusammenfassung

- ▶ Zustandsübergangssystem, Petri-Netz
- ▶ Spursprache, Omega-reguläre Wörter, Temporal-Logik
- ▶ sicherer Zugriff auf gemeinsamen Speicher:  
Semaphore, Transaktionen (STM), atomare Transaktionen (CAS)
- ▶ Prozesse mit lokalem Speicher:  
CSP, Bisimulation, Kanäle, Aktoren, verteilte Programme
- ▶ deterministischer Parallelismus:  
Strategie-Annotationen, balancierte folds, map/reduce

Beispielklausur (alt) <http://www.imn.htwk-leipzig.de/~waldmann/edu/ss11/skpp/klausur/>