Sprachkonzepte der Parallelen Programmierung Vorlesung Wintersemester 2011

Johannes Waldmann, HTWK Leipzig

29. Juni 2011

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

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

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

  Haskell: MVar, Chan
- lokale Parameterübergabe zwischen Co-Routinen

  Scala: Actor, Ada: Rendezvous

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/parallel/3.1.0.1/doc/html/Control-Parallel.html`

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/MapReduce/`

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

Data Parallelism

Idee: Elemente von Datenstrukturen automatisch (ohne Angabe einer Strategie) parallel auswerten.

Beispiele

- Vektoren (flat data parallelism)
- Bäume (nested data parallelism)

Bsp. Primzahlsieb

`http://www.haskell.org/haskellwiki/GHC/Data_Parallel_Haskell`

Threads, Thread-Sicherheit

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)

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

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.

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)

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).

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)

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)

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.

Warteschlangen

Wie synchronisiert Threads über einen Zeitraum, der länger ist als ein Methoden-Aufruf?

In Java besitzt jedes Objekt eine Warteschlange (*wait set*) von Threads.

- `ob.wait()` : der aktuelle Thread wartet (blockiert), d. h. wird in die Warteschlange von `ob` aufgenommen,
- `ob.notify()` : ein beliebiger der Threads aus der Warteschlange von `ob` wird aufgeweckt.

für jede Methode muß man den Objekt-Lock besitzen:

- `wait()` gibt den Lock frei
- der durch `notify()` aufgeweckte Thread erhält den Lock zurück.

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)

Petri-Netze

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

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
- Nachbereich (Ausgänge) einer Transition

PS: für wer nur das Skript liest: hier fehlen exakte Definitionen, diese werden in der Vorlesung an der Tafel entwickelt.

Zustände, Übergänge

- *Zustand* eines Netzes N ist Abbildung $z : S \to \mathbb{N}$

  (für jede Stelle eine Anzahl von Marken)

- in Zustand z ist eine Transition t *aktiviert*, wenn jede Stelle ihres Vorbereiches wenigstens eine Marke enthält

- eine aktivierte Transition *schaltet*: verbraucht Marken im Vorbereich, erzeugt Marken im Nachbereich.

  das ist eine Relation auf Zuständen $z_1 \to_t z_2$: aus Zustand z_1 wird durch Schalten von t der Zustand z_2 erreicht.

- zum Netz sollte auch Anfangszustand z_0 gegeben sein. Definition: erreichbare Zustände, tote Zustände

Petri-Netze modellieren. . .

- sequentielle Ausführung
- Auswahl (ein Zweig von mehreren)
- nebenläufige Verzweigung (mehrere Zweige)
- Synchronisation
- Konflikte (gegenseitiger Ausschluß)

Sprache eines Netzes

- abstrahiere von Folge von Zuständen $z_0 \to_{t_1} z_1 \to_{t_2} z_2 \ldots$

  zur *Spur*: Folge der dabei geschalteten Transitionen $t_1 t2 \ldots$

  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

Bedingung/Ereignis-Netze

. . . erhält man aus allgemeinem Modell durch:

- jede Kante hat Gewicht 1
- jede Kapazität ist 1

Beispiele:

- Ampelkreuzung (zwei Ampeln grün/gelb/rot, nicht gleichzeitig grün)
- speisende Philosophen

  Analyse von Lebendigkeit, Fairness

Transitionen in Java

verwendet zur Synchronisation von Threads:

- `cb = CyclicBarrier (int n)`

  entspricht einer Transition mit n Eingängen und n Ausgängen,

- die Stellen sind die Threads vor und nach `cb.await()`

Anwendung Partikelsimulationen, evolut. Optimierung usw.:

- pro Generation viele unabhängige Teilrechnungen,
- bei Übergang zu nächster Generation: Synchronisation

vergleiche CUDA: `__synchthreads()`

Lokale Prozeßkommunikation

Motivation

bisher betrachtete Modelle zur Thread-Kommunikation/Verwaltung:

- Datenaustausch über gemeinsamen Speicher
- Synchronisation durch Locks

jetzt: höhere Abstraktionen zum Datenaustausch

- lokal (kein globaler Speicher)
- automatisch synchronisiert

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

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

Modellierung mittels Petri-Netz?

Rendezvous (III)

allgemeinere Formen von `accept`:

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

  Modellierung mittels Petri-Netz?

- ```
  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://www.adaic.org/resources/add_content/standards/05aarm/html/AA-9-7-1.html`

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
```

Kommunikations-Kanäle

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

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

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`

Communicating Sequential Processes

Einleitung

CSP $=$ abstraktes Modell für Kommunikation von Prozessen

C. A. R. Hoare 1978,

Grundlage für Prozeßmodell in Occam, Ada, Go, ...

Definition (stark vereinfacht):

CSP $=$ (endliche) Automaten, genauer betrachtet

- Zustand $=$ Zustand
- Alphabet $=$ atomare Ereignisse
- Sprach-Äquivalenz $\not\Rightarrow$ Prozeß-Äquivalenz

Die Prozeß-Algebra (Syntax)

Terme zur Beschreibung von Prozessen
(vgl. reguläre Ausdrücke zur Beschreibung von Sprachen)
Menge $\mathrm{Proc}(\Sigma)$ der Prozeß-Terme über einem Alphabet Σ von atomaren Aktionen:

- $\mathrm{STOP} \in \mathrm{Proc}(\Sigma)$
- Präfix: $a \in \Sigma \wedge P \in \mathrm{Proc} \Rightarrow (a \to P) \in \mathrm{Proc}$
- external choice: $P_i \in \mathrm{Proc}(\Sigma) \Rightarrow (P_1 \Box P_2) \in \mathrm{Proc}(\Sigma)$
- internal choice: $P_i \in \mathrm{Proc}(\Sigma) \Rightarrow (P_1 \sqcap P_2) \in \mathrm{Proc}(\Sigma)$

und weitere Operatoren (später)

- Fixpunkt (für Iteration)
- Verschränkung (für nebenläufige Ausführung)

Die Prozeß-Algebra (operationale Semantik)

durch Inferenz-System ($=$ Axiome, Schlußregeln)
für die Übergangsrelation zwischen Prozessen

$P_1 \xrightarrow{a} P_2$ bedeutet für $a \in \Sigma$: Prozeß P_1 führt a aus (sichtbar) und verhält sich danach wie Prozeß P_2

$P_1 \xrightarrow{\tau} P_2$ bedeutet (mit $\tau \notin \Sigma$) Prozeß P_1 führt eine nicht sichtbare Aktion aus und verhält sich danach wie Prozeß P_2

- Axiom $(a \to P) \xrightarrow{a} P$
- Axiome $(P_1 \sqcap P_2) \xrightarrow{\tau} P_1$ und $(P_1 \sqcap P_2) \xrightarrow{\tau} P_2$
- Schlußregel:

  wenn $P \xrightarrow{a} Q$, dann $(P \,\square\, P') \xrightarrow{a} Q$ und $(P' \,\square\, P) \xrightarrow{a} Q$

Sequentielle Komposition

- Syntax $P_1; P_2$
- Semantik
  - informal: erst P_1, dann P_2
  - formal (Inferenz-Regel)

Nebenläufige Komposition

... mit Synchronisation

- Syntax: $(P_1 \|_S P_2)$ für $S \subseteq \Sigma$
- Semantik
  - informal: Ereignisse aus S
    sollen in P_1 und P_2 synchron ausgeführt werden,
    Ereignisse $\notin S$ beliebig (asynchron)
  - formal (Inferenzregeln): ...

beachte Spezialfälle $S = \emptyset, S = \Sigma$.

Iteration (Rekursion)

- Syntax: $\mu P.(\underbrace{\ldots P \ldots}_{Q})$

  P wird definiert durch einen Ausdruck Q, der P enthält

- Semantik (Inferenz-Regel)

  wenn $Q[P := (\mu P.Q)] \stackrel{a}{\to} R$,

  dann $(\mu P.Q) \stackrel{a}{\to} R$

Endliche Systeme

für $P \in \mathrm{Proc}(\Sigma)$: Menge der von P aus durch Folgen von a– und τ–Schritten erreichbaren Prozesse (Terme) ist endlich,

- die meisten Inferenzeregeln erzeugen nur echte Teilterme
- beachte: Komposition $\|_S$
- beachte: Fixpunkt-Operator (Rekursion)

  $\mu P.(\ldots P \ldots)$

einfache Lösung: Einschränkung auf End-Rekursion
$\mu P.(\ldots P)$
vergleiche: rechtslineare Grammatiken

Spur-Semantiken

bis jetzt: Zustandsübergangssystem

- Zustände $=$ Prozeß-Terme,
- Übergänge $\stackrel{a}{\to}$ und $\stackrel{\tau}{\to}$

entspricht: (endlicher) Automat mit Epsilon-Übergängen

jetzt: Vergleich solcher Systeme (Automaten) durch Beobachtungen.

- (bekannt) Aktionen sind beobachtbar

  denotationale Semantik des System ist Spur-Sematik: Menge aller möglichen Aktionsfolgen

- (neu) Deadlocks sind beobachtbar, Verfeinerung der Spur-Semantik

Spur/Ablehnungs-Semantik

Spur:

- Ein-Schritt-Relation: ...
- Mehr-Schritt-Relation: ...

Ablehnung:

- $P \in \mathrm{Proc}(\Sigma)$ heißt *stabil*, wenn $\neg\exists Q \in \mathrm{Proc}(\Sigma) : P \xrightarrow{\tau} Q$
- stabiler $P \in \mathrm{Proc}(\Sigma)$ *lehnt* $A \subseteq \Sigma$ *ab*,
  wenn $\neg\exists a \in A, Q \in \mathrm{Proc}(\Sigma) : P \xrightarrow{a} Q$

Ablehnungs-Semantik $\mathrm{rej}(P)$ Menge aller Paare von Spur und Ablehnungsmenge am Ende der jeweiligen Spur.

Divergenz

Motivation:

- ein Prozeß *divergiert*, wenn er unendliche viele τ-Übergänge direkt nacheinander ausführt
- ein divergenter Prozeß hat keine sequentielle Fortsetzung
- durch $P; Q$ kann man „beobachten," ob P divergiert

Semantik-Begriff erweitern:
$\mathrm{div}(P) =$ Menge aller Spuren, die von P zu einem Prozeß führen, der divergiert.

CSP-Literatur

S. Brookes, A. W. Roscoe, and D. J. Walker: *An Operational Semantics for CSP.* Manuscript, 1988.

Text: `http://www.cs.cmu.edu/afs/cs.cmu.edu/user/brookes/www/papers/OperationalSemanticsCSP.pdf`

Autor: `http://www.cs.cmu.edu/afs/cs.cmu.edu/user/brookes/www/`

Ableitungen (1)

(algebraische Methoden, angewendet auf reguläre Ausdrücke)

Analogie zwischen

- partieller Ableitung $\frac{\partial xy}{\partial x} = y$
- Prozeß-Übergang $(x \to (y \to \mathrm{STOP})) \xrightarrow{x} (y \to \mathrm{STOP})$

Übung: Ableitung einer Summe? eines Produktes?

Ableitungen (2)

definiere Relation $X_1 \xrightarrow{a} X_2$
für Ausdrücke X_1, X_2 und Zeichen $a \in \Sigma$

bestimme alle von X durch $\xrightarrow{a}$-Pfade erreichbaren Ausdrücke.
der so definierte Graph ist ein endlicher Automat für $\mathrm{Lang}(X)$

man erhält ein Verfahren zur Automaten-Synthese:

- Eingabe: regulärer Ausdruck X
- Ausgabe: endlicher Automat A mit $\mathrm{Lang}(A) = \mathrm{Lang}(X)$

und A ist sogar vollständig und deterministisch.

Ableitungen (3)

Syntax (einfacher) regulärer Ausdrücke:

- elementar reguläre Ausdrücke

  Buchstabe, leeres Wort, leere Menge

- zusammengesetzte reguläre Ausdrücke

  Vereinigung, Verkettung, Wiederholung

definiere durch Rekursion über diese Syntax

- Prädikat $E(X)$, Spezifikation: $E(X) \iff \epsilon \in \mathrm{Lang}(X)$
- Nachfolger $\frac{\partial X}{\partial a}$,

  Spezifikation: $\mathrm{Lang}(\frac{\partial X}{\partial a}) = \{w \mid a \cdot w \in \mathrm{Lang}(X)\}$

Ableitungen (4)

- Ableitung eines Buchstabens

  $\frac{\partial b}{\partial a} =$ (wenn $a = b$ dann ϵ sonst $\emptyset$)

- Ableitung des leeren Wortes

  $\frac{\partial \epsilon}{\partial a} = \emptyset$

- Ableitung der leeren Menge

  $\frac{\partial \emptyset}{\partial a} = \emptyset$

Ableitungen (5)

- Ableitung einer Vereinigung:

  $\frac{\partial(X+Y)}{\partial a} = \frac{\partial X}{\partial a} + \frac{\partial Y}{\partial a}$

- Ableitung einer Verkettung: $\frac{\partial(X \cdot Y)}{\partial a} = \ldots$
  - wenn $\neg E(X)$, dann $\cdots = \frac{\partial X}{\partial a} \cdot Y$
  - wenn $E(X)$, dann $\cdots = \frac{\partial X}{\partial a} \cdot Y + \frac{\partial Y}{\partial a}$

- Ableitung einer Wiederholung:

  $\frac{\partial(X^*)}{\partial a} = \frac{\partial(X)}{\partial a} \cdot X^*$

Vereinfachungsregeln für ϵ und $\emptyset$ als Argumente für $+$ und $\cdot$

Beschränkte Rekursion

beliebige Rekursion kann zu unendlichen Zustandsgraphen und nicht-regulären (Spur-)Sprachen führen.

Falls das nicht gewünscht ist, dann Syntax einschränken:

- $\mu P.((a \to P; c)\Box b)$

  Spursprache: ...

  Folgerung: Einschränken auf rechts-lineare Ausdrücke (P darf niemals links von `;` stehen)

- Simulation von `;` durch $\|_S$:

  $\mu P.((a \to P\|_a ac)\Box b)$

  Folgerung: P darf gar nicht in $\|_S$ vorkommen

Bisimulation von Prozessen

Plan

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

Semantiken und durch sie definierte Äquivalenzen:

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

Definition

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

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

- Vor- und Nachbereich groß genug:
  $\operatorname{domain} R = Q_1, \operatorname{range} R = Q_2$
- Startzustände sind bisimilar: $(i_1, i_2) \in R$
- S_1-Transitionen durch S_2-Transitionen simuliert:
  $\forall (p_1, p_2) \in R : \forall (p_1, a, q_1) \in T_1 :$
  $\quad \exists q_2 : (p_2, a, q_2) \in T_2 \wedge (q_1, q_2) \in R$
- 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 $R_0, R_1 \ldots \subseteq Q_1 \times Q_2$ 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: $Q_1 \times Q_2 = R_0 \supseteq R_1 \supseteq R_2 \supseteq \ldots$
- 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 $1 \leftrightarrow 2$)

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)

Literatur

- Kapitel 15 aus: Brian Goetz et al.: *Java Concurrency in Practice*
- Which CPU architectures support Compare And Swap (CAS)?

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

Compare-and-Set (Benutzung)

```
AtomicInteger p;

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

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:+PrintAssembly CAS
```

`http://wikis.sun.com/display/`
`HotSpotInternals/PrintAssembly`

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

auch nützlich: `http://blogs.sun.com/watt/`
`resource/jvm-options-list.html`

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/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

Das ABA-Problem

compare-and-set für Verweise:

Verweise könnten übereinstimmen, aber Bedeutungen nicht (Beispiel: bei selbstgebauter Speicherverwaltung—ein korrekter Gargabe Collector würde den alten Verweis mit ändern)

Ausweg: Verweise mit Versionsnummern (AtomicStampedReference)

Elimination Tree Stack

Nir Shivat: Data Structures in the Multicore Age, Communications ACM, 03/2011.

Plan:

- Ausgangspunkt: lock-free Stack
- Elimination Array
- Baum von Stacks
- Elimination Array in jedem Knoten

Semantik von nebenläufigen Stacks

- Dyck-Sprache, gegeben durch Grammatik mit Regeln $S \to \epsilon, S \to \text{Push}(i)\ S\ \text{Pop}(i)\ S$
- Datenstruktur ist (klassischer) Stack $\iff$ Spursprache ist Präfixsprache der Dyck-Sprache
- Nebenläufigkeit: Operationen $\text{Push}_k(i), \text{Pop}_k(i)$

  Op. mit unterschiedlichem Index sind vertauschbar
- nebenläufiger Stack: jede Spur ist vertauschungs-äquivalent zu Präfix eines Dyck-Wortes
- später Abschwächung: betrachte nur “ruhende” Spuren (kein Thread blockiert eine Ressource)

Elimination Array

- die Threads, die push/pop nicht ausführen können (weil andere Threads zugreifen und deswegen CAS fehlschlägt),
- belegen während des Wartens eine zufällige Position in einem Eliminiation Array.
- wenn sich dort Threads mit push und pop treffen, sind diese Threads fertig und brauchen den Stack gar nicht.

Baum von Stacks

- vollständiger binärer Baum
- jedes Blatt ist lock-free stack
- jeder innere Knoten enthält ein Bit,
  Bit 0/1 $=$ Richtung Links/Rechts

Operationen:

- push: umschalten (CAS) und in die alte Richtung gehen
- pop: umschalten (CAS) und in die neue Richtung gehen

Eigenschaften:

- Stacktiefen sind balanciert
- Pop nach Push: aus gleichem Stack

Baum mit Eliminatoren

- Flaschenhals ist das Bit in der Wurzel.
- Eliminator-Array in jedem Knoten
  - push/pop: sofort eliminieren
  - push/push und pop/pop:
    auf Teilbäume verteilen, ohne Bit zu lesen/umzuschalten!

nebenläufige Stack-Semantik jetzt nur noch für ruhende Spuren ($=$ wenn kein Thread im Baum wartet)
Ü: Beispiel

Software Transactional Memory

Einleitung

Benutzung:

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

Literatur zu STM

- 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/`
- Stuart Halloway: *Concurrency*, = Kapitel 6 in: *Programming Clojure*, The Pragmatic Bookshelf, 2009.
- Scala STM Expert Group, `http://nbronson.github.com/scala-stm/`

  "We're working on a lightweight software transactional memory for Scala, inspired by the STMs in Haskell and Clojure. Our target is inclusion in Scala's standard library for 2.9.1. "

- A. Dragojevic, P. Felber, V. Gramoli, R. Guerraoui: *Why STM can be more than a research toy*, in: Communications ACM, 4/2011.

  “despite earlier claims, Software Transactional Memory outperforms sequential code.”

Plan

- STM in Haskell (Beispiele)
- Haskell-Typsystem:
  - zur Absicherung von STM
  - Haskell brauchen wir später nochmal
- STM in Clojure und/oder Scala

Beispiel (ohne STM)

```
main = do
  let steps = 10000
      threads = GHC.Conc.numCapabilities
  counter <- newIORef 0
  ch <- newChan -- als Latch benutzt
  forM [ 1 ..  threads ] $ \ t -> forkIO $ do
      forM [ 1 .. steps ] $ \ s -> do
          c <- readIORef counter
          writeIORef counter (c + 1)
      writeChan ch ()
  forM [ 1 .. threads ] $ \ s -> readChan ch
  c <- readIORef counter
  putStrLn $ show c
```

Beispiel (mit STM)

```
main = do
  let steps = 10000
      threads = GHC.Conc.numCapabilities
  counter <- atomically $ newTVar 0
  ch <- newChan
  forM [ 1 ..  threads ] $ \ t -> forkIO $ do
    forM [ 1 .. steps ] $ \ s -> atomically $ do
        c <- readTVar counter
        writeTVar counter (c + 1)
    writeChan ch ()
  forM [ 1 .. threads ] $ \ s -> readChan ch
  c <- atomically $ readTVar counter
  putStrLn $ show c
```

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`

Philosophen mit STM

kein Deadlock (trivial).

```
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
```

nicht fair.

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) ...
(printf 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

Ameisen

Wegfindung (Nahrungssuche) gesteuert durch Pheromone.

Zustandsänderungen (Ameise, Spielfeld) synchronisiert durch Transaktionen.

ist beliebter Testfall für nebenläufige Programme:

- Rich Hickey, Clojure: `http://clojure.googlegroups.com/web/ants.clj`
- Peter Vlugter, Scala: `https://github.com/jboner/akka/tree/master/akka-samples/akka-sample-ants/`
- Jeff Foster, Haskell: `http://www.fatvat.co.uk/2010/08/ants-and-haskell.html`

Ameisen (Modell)

```
type Pos = (Int, Int)
data Ant = Ant { position :: TVar Pos
               , direction :: TVar Int
               }
data Cell = Cell { occupied :: TVar Bool
                 , pheromone :: TVar Double
                 }
type Board = Array Pos Cell
data World = World { board :: Board
             , population :: [ Ant ]
             }
```

- Invariante?

- Zustandsübergänge?

Ameisen (Aufgabe)

1. Ameise darf nur auf leeres Nachbarfeld laufen.

- Transaktionen für: Feld verlassen, Feld betreten

2. Ameise soll ein zufälliges solches Nachbarfeld betreten.

- Welcher Fehler ist in folgendem Ansatz?

```
walk :: World -> Ant -> IO ()
walk w a = do
    d <- random ( 0, 7 )
    atomically $ do
        a läuft_in_Richtung d
```

Ameisen (Haskell) (Vorbereitung)

```
import Data.Array
import Control.Concurrent.STM
import Control.Concurrent
import Control.Monad ( forM, forever, void )
import System.Random
import System.IO

type Pos = (Int,Int)
type Dir = Int

data Ant = Ant { position :: TVar Pos
               , direction :: TVar Dir
               , moves :: TVar Int
```

```
                }

data Cell = Cell { occupied :: TVar Bool
                  }
type Board = Array Pos Cell

data World = World
             { size :: Pos
             , board :: Board
             , population :: [ Ant ]
             , generator :: TVar StdGen
             }

-------------------------------------------------------------------
```

```
main :: IO ()
main = do
    w <- make_world (20,20) 10
    forM ( population w ) $ \ ant ->
        forkIO $ forever $ walk w ant
    forever $ do
        pos <- snapshot w
        threadDelay $ 10^6 -- microseconds

info :: Ant -> STM String
info ant = do
    pos <- readTVar $ position ant
    dir <- readTVar $ direction ant
    mov <- readTVar $ moves ant
```

```
    return $ unwords [ "pos", show pos, "dir", show dir, "moves", show mov ]

snapshot :: World -> IO ()
snapshot w = do
  infos <- atomically $ forM ( population w ) $ info
  putStrLn $ unlines infos

---------------------------------------------------------------------------

-- | verschiebe in gegebene Richtung,
-- mit wrap-around an den Rändern (d.h. Torus)
shift :: (Int,Int) -> Pos -> Dir -> Pos
shift (width,height) (x,y) d =
    let (dx,dy) = vector d
```

```
    in  ( mod (x+dx) width, mod (y+dy) height )

vector :: Dir -> Pos
vector d = case mod d 8 of
    0 -> ( 1,0) ; 1 -> ( 1, 1) ; 2 -> (0, 1) ; 3 -> (-1, 1)
    4 -> (-1,0) ; 5 -> (-1,-1) ; 6 -> (0,-1) ; 7 -> ( 1,-1)

----------------------------------------------------------------

randomRT :: Random a => TVar StdGen -> (a,a) -> STM a
randomRT ref bnd = do
    g <- readTVar ref
    let (x, g') = randomR bnd g
    writeTVar ref g'
```

```
    return x

random_selection :: TVar StdGen -> Int -> [a] -> STM [a]
random_selection ref 0 xs = return []
random_selection ref k xs = do
    ( pre, y : post ) <- random_split ref xs
    ys <- random_selection ref (k-1) ( pre ++ post )
    return $ y : ys

random_split :: TVar StdGen -> [a] -> STM ( [a], [a] )
random_split ref xs = do
    k <- randomRT ref ( 0, length xs - 1 )
    return $ splitAt k xs
```

```
------------------------------------------------------------------

make_world :: (Int,Int) -> Int -> IO World
make_world (width,height) num_ants = do
    b <- make_board (width, height)
    gen <- newStdGen
    ref <- atomically $ newTVar gen
    ants <- make_ants ref b num_ants
    return $ World
        { size = (width, height), board = b, population = ants
        , generator = ref
        }

make_board :: (Int,Int) -> IO Board
make_board (width,height) = do
```

```
    let bnd = ((0,0),(width-1,height-1))
    cells <- forM ( range bnd ) $ \ xy -> do
          occ <- atomically $ newTVar False
          return (xy, Cell { occupied = occ } )
    return $ array bnd cells

make_ants :: TVar StdGen -> Board -> Int -> IO [ Ant ]
make_ants ref b num_ants = atomically $ do
    sel <- random_selection ref num_ants $ indices b
    forM sel $ \ pos -> do
        p <- newTVar pos
        enter $ b ! pos
        dir <- randomRT ref ( 0, 7 )
        d <- newTVar dir
```

```
m <- newTVar 0
return $ Ant { position = p, direction = d, moves = m }
```

Ameisen (Haskell) (Aufgaben)

Ergänzen Sie die Funktionen (Transaktionen):

```
count :: Ant -> STM ()
count ant = do
    m <- readTVar ( moves ant )
    writeTVar ( moves ant ) $! (m + 1 )

-- | Zelle betreten, falls sie frei ist
enter :: Cell -> STM ()
enter c = do
    ... <- readTVar ...
    check ...
    writeTVar ...
-- | Zelle verlassen
```

```
leave :: Cell -> STM ()

-- | ein Schritt in aktuelle Richtung
forward :: World -> Ant -> STM ()
forward w ant = do
    ...
    let pos’ = shift ( size w ) pos dir
    ...
-- | Richtung verändern
rotate :: Int -> Ant -> STM ()
rotate turn ant = do
    ...
    ...    ( dir + turn )
```

Ameisen (Haskell) (Diskussion)

verschiedene Varianten für das Laufen einer Ameise:

- Drehung vor dem Laufen würfeln (separate Transaktionen)

```
walk :: World -> Ant -> IO ()
walk w ant = do
    turn <- atomically
         $ randomRT ( generator w ) (-1, 1)
    atomically $ do
        rotate turn ant
        forward w ant
        count ant
```

- Drehung innerhalb der Lauf-Transaktion würfeln

- welches ist eine passende Ersatz-Transaktion, falls Commit für Laufen fehlschlägt?

```
forever $ orElse (walk w ant) ...
```

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 Annotationen (Kombinatoren)
  `par X (pseq Y (f X Y))`
- Haskell benutzt (in diesem Fall: leider) Bedarfsauswertung, diese muß man ggf. umgehen, d.h. Auswertung von Teilausdrücken erzwingen

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
prime :: Int -> Bool
trial_div :: Int -> Int -> Bool
summe :: [ Int ] ->  Int
```

Verteile auf mehrere Teilsummen

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.

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

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)

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 a
  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)

$\Rightarrow$ 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 ) )
```

Anonyme Funktionen

- Syntax:

  $v_1, \ldots, v_n$ sind formale Parameter, B ein Ausdruck
  - Mathematik: $\lambda v_1 \ldots v_n.B$
  - Haskell: `\ v1 .. vn -> B`
  - C#: `(v1, .., vn) => B`
- Semantik: Funktion auf Argumente anwenden: x

  $(\lambda v_1 \ldots v_n.B)A_1 \ldots A_n$

  in B wird jedes v_i durch (Wert von) A_i ersetzt.

  (Genaueres dazu später im Compilerbau.)
- Beispiele:
  - $(\lambda fxy.fyx)\ (\lambda ab.a - b)\ 1\ 2$
  - $(\lambda fx.f(fx))\ (\lambda a.a + 1)\ 0$

Rekursion über Listen

```
and :: List Bool  -> Bool
and xs = case xs of
     Nil -> True ; Cons x  xs' -> x && and xs'
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));
```

Besucher-Muster für Bäume

```
interface Tree<K> { }
class Leaf<K>   implements Tree<K> {
  Leaf(K key) { .. } }
class Branch<K> implements Tree<K> {
  Branch(Tree<K> left, Tree<K> right) { .. } }
```

für jeden Teilnehmer des Kompositums eine Methode:

```
interface Visitor<K,R> { // mit Resultattyp R
    R leaf (K x);
    R branch (R left, R right);  }
```

der Gastgeber (Baumknoten) nimmt Besucher auf:

```
interface Tree<K> {
  <R> R receive (Visitor<K,R> v) }
```

Beispiel: Baum-Besucher

Benutzung des Besuchers: Anzahl der Blätter:

```
class Trees {
  static <K> int leaves (Tree<K> t) {
    return t.receive(new Tree.Visitor<K,Integer>() {
      public Integer branch
           (Integer left, Integer right) {
        return left + right;
      }
      public Integer leaf(K key) {
        return 1;
      }
    });
  }
```

```
}
```

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:

$\Sigma =$ Monoid (Eins-Element 1, binäre Operation $\circ$)

$A = \text{List a}$ (Listen) mit $1_A = \text{Nil}, \circ_A = \text{append}$

$B = \text{N}$ (Zahlen) mit $1_B = \text{Z}, \circ_A = \text{plus}$

$h = \text{length}$

Homomorphismen von Listen

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, `fold`: balancierter Binärbaum.

2. jeder Hom. kann als foldl und als foldr geschrieben werden

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

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.

foldr, foldl, reduce

- Rechnung beginnt am rechten Ende, entspricht dem natürlichen Rekursionsschema:

```
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 :: (a -> a -> a) -> [a] -> a
foldb :: (a -> b)
      -> (b -> b -> b) -> [a] -> b
```

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' xs = ...`
- 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`

Intel Manycore Testing Lab

`http://software.intel.com/en-us/articles/intel-many-core-testing-lab/`

- kostenloses, zeitlich beschränktes Angebot für Lehre:
- Benutzung von Maschinen mit Intel-Prozessoren mit vielen Kernen (32, 40), viel Speicher (256 GB), Standardsoftware (Compiler C, Java, Haskell)
- Studenten-Accounts, Zugriff nur aus HTWK-Netz. Benutzerordnungen (Intel, HTWK). Datenschutz (keine Garantien, aber Anonymität gegenüber Intel)
- ein zentraler Knoten (zum Kompilieren und Testen), mehrere Batch-Knoten (zum Rechnen), keine Internet-Verbindung nach außen

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

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

`http://www.imn.htwk-leipzig.de/~waldmann/ss11/skpp/code/kw24/mps-vector.hs`

zusätzl. Informationen:

- Vektoren:
  `http://hackage.haskell.org/package/vector`
- effiziente Code-Erzeugung (Inlining)
  `http://article.gmane.org/gmane.comp.lang.haskell.cafe/90211`
- paper folding sequence: `http://oeis.org/A014577`

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.html`

- 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-courses/hs08/map-reduce/reading/mapreduce-progmodel-scp08.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

```
map_reduce :: ( Ord ki, Ord ko )
  => ( (ki, vi) -> [(ko,vm)] )  -- ^ distribute
  -> ( ko -> [vm] -> Maybe vo ) -- ^ collect
  -> Map ki vi -- ^ eingabe
  -> Map ko vo -- ^ ausgabe
map_reduce distribute collect input
    = M.map ( \ ( Just x ) -> x )
    $ M.filter isJust
    $ 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 context) { } }
public static class IntSumReducer
   extends Reducer<Text,IntWritable,Text,IntWritable> {
   public void reduce(Text key, Iterable<IntWritable> values, Context context ) { } }
 }
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 `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`

Data Parallelism

RePA (regular parallel arrays)

`http://repa.ouroborus.net/`

Beispiel: Matrixmultiplikation

Parallel Linq

http://msdn.microsoft.com/en-us/library/dd460688.aspx

```
var source = Enumerable.Range(1, 10000);
var evenNums = from num in source.AsParallel()
               where Compute(num) > 0
               select num;
```

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

Zusammenfassung

- Petri-Netz, Zustandsübergangssystem,
- Kanal, MVar, Agent/Actor
- software transactional memory (TVar)
- parallele Auswertung funktionaler Programme (par, pseq)
- balancierte folds, 3. Homomorphiesatz
- map/reduce