Fortgeschrittene Techniken der Funktionalen Programmierung
Vorlesung vom 15.12.09:
Nebenläufigkeit in Haskell: Software Transactional Memory

Christoph Lüth, Dennis Walter

Universität Bremen

Wintersemester 2009/10

Fahrplan

- Teil I: Monaden und fortgeschrittene Typen
- Teil II: Fortgeschrittene Datenstrukturen
- Teil III: Nebenläufigkeit
  - Grundlagen
  - Abstraktionen und Ausnahmebehandlung
  - Software Transactional Memory
- Teil IV: The Future of Programming

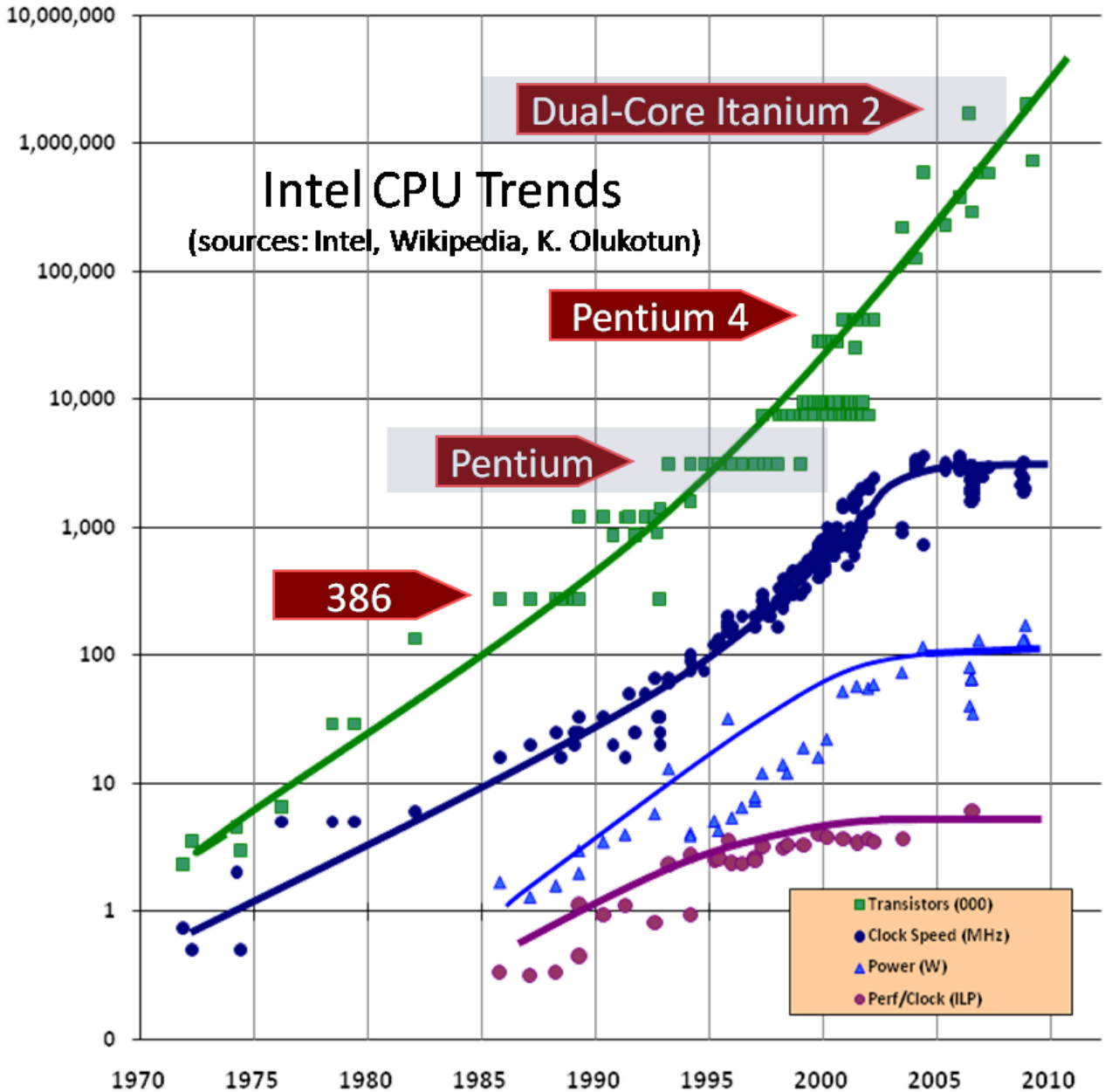
Heute gibt es:

- Einen fundamenal anderen Ansatz nebenläufiger Datenmodifikation
  - Keine Locks und Conditional variables
  - Sondern: Transaktionen!
  - Software transactional memory (STM)
- Implementierung in Haskell: `atomically`, `retry`, `orElse`
- Fallbeispiele:
  - Puffer: Reader-/Writer
  - Speisende Philosophen (nur im Code: `MySTM.hs`)
  - Weihnachtlich: das Santa Claus Problem

Die Nebenläufigkeitsrevolution

- Free lunch is over: CPUs werden nicht mehr schneller
  - 2GHz hatten wir schon 2001!
  - Heute: $\sim$3GHz
- Dafür vermehren sie sich plötzlich!
  - Quad-Core, Octo-Core, Cluster, ...
- Die großen Chip-Hersteller arbeiten an Mehr{Kern, CPU, Rechner}-Lösungen, um Moore's Law nicht zu verletzen
  - Intel Core™2, AMD Phenom™, nVidia GeForce Dual-Chip (SLI™)
- Auswirkungen auf Softwareentwicklung!
  - Verstärkter Fokus auf nebenläufige SW

Schneller geht's nimmer



Locks und Conditional variables

- Aktueller “Stand der Technik”
  - C

```
pthread_mutex_lock(&mutex)
pthread_mutex_unlock(&mutex)
pthread_cond_wait(&cond, &mutex)
pthread_cond_broadcast(&cond)
```

  - Haskell

```
newMVar :: a → IO (MVar a)
takeMVar :: MVar a → IO a
putMVar :: MVar a → a → IO ()
```

  - Java

```
synchronized public void workOnSharedData() { ... }
```

Locks und Conditional variables (2)

- Grundlegende Idee: Zugriff auf gemeinsame Ressourcen nur innerhalb kritischer Abschnitte
  1. Vor Betreten um Erlaubnis fragen (Lock an sich reißen)
  2. Arbeiten
  3. Beim Verlassen Meldung machen (Lock freigeben)

Locks und Conditional variables (2)

- Grundlegende Idee: Zugriff auf gemeinsame Ressourcen nur innerhalb kritischer Abschnitte
  1. Vor Betreten um Erlaubnis fragen (Lock an sich reißen)
  2. Arbeiten
  3. Beim Verlassen Meldung machen (Lock freigeben)
- Verfeinerung: Auf Eintreten von Bedingungen warten (Kommunikation)
  1. Im kritischen Abschnitt schlafengehen, wenn Bedingung nicht erfüllt (Lock freigeben!)
  2. Andere Threads machen Bedingung wahr und melden dies
  3. Sobald Lock verfügbar: aufwachen

Locks und Conditional variables (2)

- Grundlegende Idee: Zugriff auf gemeinsame Ressourcen nur innerhalb kritischer Abschnitte
  1. Vor Betreten um Erlaubnis fragen (Lock an sich reißen)
  2. Arbeiten
  3. Beim Verlassen Meldung machen (Lock freigeben)
- Verfeinerung: Auf Eintreten von Bedingungen warten (Kommunikation)
  1. Im kritischen Abschnitt schlafengehen, wenn Bedingung nicht erfüllt (Lock freigeben!)
  2. Andere Threads machen Bedingung wahr und melden dies
  3. Sobald Lock verfügbar: aufwachen
- Semaphoren & Monitore bauen essentiell auf demselben Prinzip auf

Kritik am Lock-basierten Ansatz

- Kritische Abschnitte haben eine pessimistische Lebenseinstellung:
  - Möglicherweise will ein anderer Thread gerade dieselben Daten verändern
  - Darum: Sperrung des Abschnitts in jedem Fall
  - Möglicherweise gar nicht nötig: Effizienz?

Kritik am Lock-basierten Ansatz

- Kritische Abschnitte haben eine pessimistische Lebenseinstellung:
  - Möglicherweise will ein anderer Thread gerade dieselben Daten verändern
  - Darum: Sperrung des Abschnitts in jedem Fall
  - Möglicherweise gar nicht nötig: Effizienz?
- Gefahr des Deadlocks:
  - A betritt kritischen Abschnitt S_1; gleichzeitig betritt B S_2
  - A will nun S_2 betreten, während es Lock für S_1 hält
  - B will dasselbe mit S_1 tun.
  - The rest is silence...

Kritik am Lock-basierten Ansatz

- Kritische Abschnitte haben eine pessimistische Lebenseinstellung:
  - Möglicherweise will ein anderer Thread gerade dieselben Daten verändern
  - Darum: Sperrung des Abschnitts in jedem Fall
  - Möglicherweise gar nicht nötig: Effizienz?
- Gefahr des Deadlocks:
  - A betritt kritischen Abschnitt S_1; gleichzeitig betritt B S_2
  - A will nun S_2 betreten, während es Lock für S_1 hält
  - B will dasselbe mit S_1 tun.
  - The rest is silence...
- Richtige Granularität schwer zu bestimmen
  - Grobkörnig: ineffizient; feinkörnig: schwer zu analysieren

Kritik am Lock-basierten Ansatz (2)

- Größtes Problem jedoch: *Lock-basierte Programme sind nicht komponierbar!*
  - Korrekte Einzelbausteine können zu fehlerhaften Programmen zusammengesetzt werden
- Klassisches Beispiel: Übertrag eines Elements von einem Container in einen anderen
  - Container-Bücherei explizit thread-safe, d.h. nebenläufiger Zugriff sicher
  - Implementierung des Übertrags:

```
transferItem item c1 c2 = do
  removeFrom c1 item
  insertInto c2 item
```

  - Problem: Zwischenzustand, in dem item in keinem Container ist
  - Plötzlich doch wieder Container-Locks erforderlich! Welche?

Kritik am Lock-basierten Ansatz (3)

- Ein ähnliches Argument gilt für Komposition von Ressourcen-Auswahl:
- Mehrfachauswahl in Unix/Linux/Mac OS X:
  - select () wartet auf mehrere I/O-Kanäle gleichzeitig
  - Kehrt zurück sobald mindestens einer verfügbar
- Annahme: Prozeduren foo() und bar() warten auf unterschiedliche Ressourcen(-Mengen). Z. B.

```
void foo(void) { /* ... */
  select(k, rd, wd, ed, &timeout);
}
```

- Keine Möglichkeit, foo() und bar() zu komponieren, etwa select (&foo, &bar)

Software transactional memory (atomicity)

- Ein optimistischer Ansatz zur Nebenläufigen Programmierung
- Prinzip der Transaktionen aus Datenbank-Domäne entliehen
- Kernidee: atomically (...) Blöcke werden atomar ausgeführt
  - (Speicher-)Änderungen erfolgen entweder vollständig oder gar nicht
  - Im letzteren Fall: Wiederholung der Ausführung
  - Im Block: konsistente Sicht auf Speicher
  - A(tomicity) und I(solation) aus ACID
- Damit deklarative Formulierung des Elementtransfers möglich:

```
atomically $
  do { removeFrom c1 item; insertInto c2 item }
```

Gedankenmodell für atomare Speicheränderungen

Mögliche Implementierung

- Thread T_1 im `atomically`-Block nimmt keine Speicheränderungen vor, sondern in schreibt Lese-/Schreiboperationen in Transaktions-Log
- Leseoperationen konsultieren zunächst Log
- Beim Verlassen des `atomically`-Blocks:
  1. globales Lock greifen
  2. konsistenter Speicher gelesen?
  3t. Änderungen einpflegen
  4t. Lock freigeben
  3f. Änderungen verwerfen
  4f. Lock freigeben, Block wiederholen

Gedankenmodell für atomare Speicheränderungen

Mögliche Implementierung

- Thread T_1 im `atomically`-Block nimmt keine Speicheränderungen vor, sondern in schreibt Lese-/Schreiboperationen in Transaktions-Log
- Leseoperationen konsultieren zunächst Log
- Beim Verlassen des `atomically`-Blocks:
  1. globales Lock greifen
  2. konsistenter Speicher gelesen?
  3t. Änderungen einpflegen
  4t. Lock freigeben
  3f. Änderungen verwerfen
  4f. Lock freigeben, Block wiederholen

Konsistenter Speicher

- Jede zugegriffene Speicherstelle hat zum Prüfzeitpunkt denselben Wert wie beim ersten Lesen

Beispiel: Konsistenter Speicher

```
atomically $
    do b1 ← getBalance acc1
       b2 ← getBalance acc2
       if sum [b1, b2] ≥ amnt
         then withdrawMoney amnt else ...
```

- Mögliches Problem: anderer Thread ruft withdrawMoney zwischen getBalance-Aufrufen auf
- acc1/2 nicht involviert: egal
- acc1 geändert
  - Wird bei Prüfung aller gelesenen/geschriebenen Speicherstellen bemerkt
- acc2 geändert
  - Egal! Konsistenz bezieht sich dann auf neuen Zustand

Blockieren / Warten (blocking)

- Atomarität allein reicht nicht: STM muss Synchronisation von Threads ermöglichen
- Klassisches Beispiel: Produzenten + Konsumenten:
  - Wo nichts ist, kann nichts konsumiert werden
  - Konsument wartet auf Ergebnisse des Produzenten

```
consumer buf = do
  item ← getItem buf
  doSomethingWith item
```

  - getItem sollte blockieren, wenn keine Items verfügbar

Kompositionales “Blockieren” mit `retry`

- Idee: ist notwendige Bedingung innerhalb atomically nicht erfüllt, wird Transaktion abgebrochen und erneut versucht

```
atomically $ do
  ...
  if (Buffer.empty buf) then retry else ...
```

- Sinnlos, sofern andere Threads Zustand nicht verändert haben!
- Daher: warten (auf Godot?)
  - Auf Änderung an in Transaktion gelesenen Variablen!
  - Genial: System verantwortlich für Verwaltung der Aufweckbedingung
- Keine lost wakeups, keine händische Verwaltung von Conditional variables

Auswahl (choice)

- Dritte Zutat für erfolgreiches kompositionales Multithreading: Auswahl möglicher Aktionen
- Beispiel: Event-basierter Webserver liest Daten von mehreren Verbindungen
- Kombinator orElse ermöglicht linksorientierte Auswahl (ähnlich $||$):

```
webServer = do
  ...
  news ← atomically $ orElse spiegelRSS cnnRSS
  req ← atomically $ foldr1 orElse clients
  ...
```

- Wenn linke Transaktion misslingt, wird rechte Transaktion versucht

Einschränkungen an Transaktionen

- Transaktionen dürfen nicht beliebige Seiteneffekte haben
  - Nicht jeder reale Seiteneffekt lässt sich rückgängig machen
  - **if** allElseFails **then** blackmailBossWith nudePics **else** ...
  - Seiteneffekte auf Transaktionsspeicher zu beschränken
- Ideal: Trennung statisch erzwingen
  - In Haskell: Trennung im Typsystem
  - IO-Aktionen und STM-Aktionen (Monaden)
  - STM Monade erlaubt Erzeugung und Verwendung von Referenzen (ähnlich MVar, IORef)

STM in Haskell

```
newtype STM a
instance Monad STM
atomically :: STM a → IO a
retry :: STM a
orElse :: STM a → STM a → STM a

data TVar
newTVar :: a → STM (TVar a)
readTVar :: TVar a → STM a
writeTVar :: TVar a → a → STM ()
```

Puffer mit STM: Modul `MyBuffer`

- Erzeugen eines neuen Puffers: newTVar mit leerer Liste

```
newtype Buf a = B (TVar [a])

new :: STM (Buf a)
new = do tv ← newTVar []
```

Puffer mit STM: Modul `MyBuffer` (2)

- Elemente zum Puffer hinzufügen (immer möglich):
  - Puffer lesen
  - Element hinten anhängen
  - Puffer schreiben

```
put :: Buf a → a → STM ()
put (B tv) x = do xs ← readTVar tv
```

Puffer mit STM: Modul `MyBuffer` (3)

- Element herausnehmen: Möglicherweise keine Elemente vorhanden!
  - Wenn kein Element da, wiederholen
  - Ansonsten: Element entnehmen, Puffer verkleinern

```
get :: Buf a → STM a
get (B tv) = do xs ← readTVar tv
                case xs of
                  [] → retry
                  (y:xs') → do writeTVar tv xs'
```

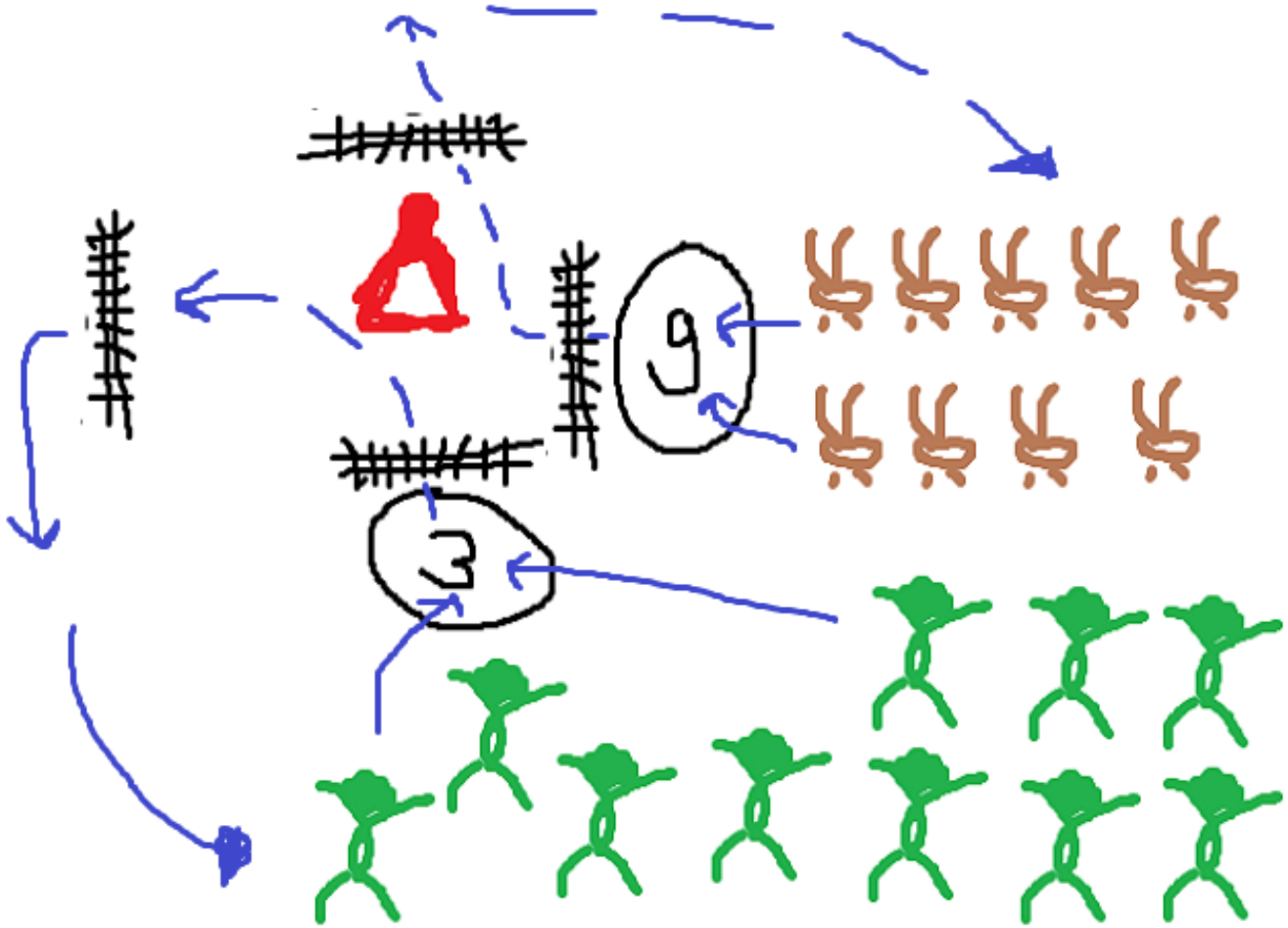
Puffer mit STM: Anwendungsbeispiel

```
useBuffer = do
  b ← atomically $ new
  forkIO $ forever $
    do threadDelay (10^6)
       t ← Tm.getClockTime >>= Tm.toCalendarTime
       atomically $ put b $
         Tm.calendarTimeToString t
  forever $ do x ← atomically $ get b
               putStrLn x
```

Santa Claus Problem

Santa repeatedly sleeps until wakened by either all of his nine reindeer, [...], 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 ([...]). If awakened by a group of elves, he shows each of the group into his study, consults with them [...], and finally shows them each out ([...]). Santa should give priority to the reindeer in the case that there is both a group of elves and a group of reindeer waiting.

Santa Claus Problem, veranschaulicht



Lösungsstrategie

- Modellieren jede Elfe, jedes Rentier, jeden Weihnachtsmann als Faden
  - Santa wartet und koordiniert, sobald genügend “Teilnehmer” vorhanden
  - Elfen und Rentiere tun fortwährend dasselbe: Sammeln, arbeiten, herumstehen

Lösungsstrategie

- Modellieren jede Elfe, jedes Rentier, jeden Weihnachtsmann als Faden
  - Santa wartet und koordiniert, sobald genügend “Teilnehmer” vorhanden
  - Elfen und Rentiere tun fortwährend dasselbe: Sammeln, arbeiten, herumstehen
- Verwenden Gruppen (Group) als Sammelplätze für Elfen und Rentiere
  - 3er-Gruppe für Elfen, 9er-Gruppe für Rentiere
  - Santa wacht auf, sobald Gruppe vollzählig

Lösungsstrategie

- Modellieren jede Elfe, jedes Rentier, jeden Weihnachtsmann als Faden
  - Santa wartet und koordiniert, sobald genügend "Teilnehmer" vorhanden
  - Elfen und Rentiere tun fortwährend dasselbe: Sammeln, arbeiten, herumstehen
- Verwenden Gruppen (Group) als Sammelplätze für Elfen und Rentiere
  - 3er-Gruppe für Elfen, 9er-Gruppe für Rentiere
  - Santa wacht auf, sobald Gruppe vollzählig
- Gatterpaare (Gate) erlauben koordinierten Eintritt in Santas Reich
  - Stellt geordneten Ablauf sicher (kein Überholen übereifriger Elfen)

Vorarbeiten: (Debug-)Ausgabe der Aktionen in Puffer

```
{- Actions of elves and deer -}
meetInStudy :: Buf → Int → IO ()
meetInStudy buf id = bput buf $
  "Elf " ++ show id ++ " meeting in the study"

deliverToys :: Buf → Int → IO ()
deliverToys buf id = bput buf $
  "Reindeer " ++ show id ++ " delivering toys"
```

- Puffer wichtig, da putStrLn nicht thread-sicher!
- Lese-Thread liest Daten aus Buf und gibt sie sequentiell an stdout aus

Arbeitsablauf von Elfen und Rentieren

- Generisch: Tun im Grunde dasselbe, parametrisiert über task

```
helper1 :: Group → IO () → IO ()
helper1 grp task = do
  (inGate, outGate) ← joinGroup grp
  passGate inGate
  task
  passGate outGate

elf1, reindeer1 :: Buf → Group → Int → IO ()
elf1 buf grp elfId =
  helper1 grp (meetInStudy buf elfId)
reindeer1 buf grp reinId =
  helper1 grp (deliverToys buf reinId)
```

Gatter: Erzeugung, Durchgang

- Gatter haben aktuelle sowie Gesamtkapazität
- Anfänglich leere Aktualkapazität (Santa kontrolliert Durchgang)

```
data Gate = Gate Int (TVar Int)

newGate :: Int → STM Gate
newGate n = do tv ← newTVar 0
               return $ Gate n tv

passGate :: Gate → IO ()
passGate (Gate n tv) =
  atomically $ do c ← readTVar tv
                  check (c > 0)
                  writeTVar tv (c − 1)
```

Nützliches Design Pattern: `check`

- Nebenläufiges assert:

```
check :: Bool → STM ()
check b | b = return ()
        | not b = retry
```

  - Bedingung b muss gelten, um weiterzumachen
  - Im STM-Kontext: wenn Bedingung nicht gilt: wiederholen
  - Nach check: Annahme, dass b gilt

- Wunderschön deklarativ!

Santas Aufgabe: Gatter betätigen

- Wird ausgeführt, sobald sich eine Gruppe versammelt hat
- Zwei atomare Schritte
  - Kapazität hochsetzen auf Maximum
  - Warten, bis Aktualkapazität auf 0 gesunken ist, d.h. alle Elfen/Rentiere das Gatter passiert haben

```
operateGate :: Gate → IO ()
operateGate (Gate n tv) = do
  atomically $ writeTVar tv n
  atomically $ do c ← readTVar tv
                  check (c ≡ 0)
```

- Beachte: Mit nur einem atomically wäre diese Operation niemals ausführbar! (Starvation)

Gruppen: Erzeugung, Beitritt

```
data Group = Group Int (TVar (Int, Gate, Gate))

newGroup :: Int → IO Group
newGroup n = atomically $ do
  g1 ← newGate n
  g2 ← newGate n
  tv ← newTVar (n, g1, g2)
  return $ Group n tv

joinGroup :: Group → IO (Gate, Gate)
joinGroup (Group n tv) =
  atomically $ do (k, g1, g2) ← readTVar tv
                  check (k > 0)
                  writeTVar tv (k − 1, g1, g2)
                  return $ (g1, g2)
```

joinGroup

- Noch einmal zum Staunen:

```
atomically $ do (k, g1, g2) ← readTVar tv
                check (k > 0)
                writeTVar tv (k − 1, g1, g2)
                return $ (g1, g2)
```

- Nebenläufigkeit ist also schwierig?!

Eine Gruppe erwarten

- Santa erwartet Elfen und Rentiere in entspr. Gruppengröße
- Erzeugt neue Gatter für nächsten Rutsch
  - Verhindert, dass Elfen/Rene

```
awaitGroup :: Group → STM (Gate, Gate)
awaitGroup (Group n tv) = do
  (k, g1, g2) ← readTVar tv
  check (k ≡ 0)
  g1' ← newGate n
  g2' ← newGate n
  writeTVar tv (n, g1', g2')
  return (g1, g2)
```

Elfen und Rentiere

- Für jeden Elf und jedes Rentier wird ein eigener Thread erzeugt
- Bereits gezeigte elf1, reindeer1, gefolgt von Verzögerung (für nachvollziehbare Ausgabe)

```
-- An elf does his elf thing, indefinitely.
elf :: Buf → Group → Int → IO ThreadId
elf buf grp id = forkIO $ forever $
  do elf1 buf grp id
     randomDelay

-- So does a deer.
reindeer :: Buf → Group → Int → IO ThreadId
reindeer buf grp id = forkIO $ forever $
  do reindeer1 buf grp id
     randomDelay
```

Santa Claus' Arbeitsablauf

- Gruppe auswählen, Eingangsgatter öffnen, Ausgang öffnen
- Zur Erinnerung: operateGate "blockiert", bis alle Gruppenmitglieder Gatter durchschritten haben

```
santa :: Buf → Group → Group → IO ()
santa buf elves deer = do
  (name, (g1, g2)) ← atomically $
    chooseGroup "reindeer" deer `orElse`
      chooseGroup "elves" elves
  bput buf $ "Ho, ho, my dear " ++ name
  operateGate g1
  operateGate g2

chooseGroup :: String → Group →
               STM (String, (Gate, Gate))
chooseGroup msg grp = do
  gs ← awaitGroup grp
  return (msg, gs)
```

Hauptprogramm

- Gruppen erzeugen, Elfen und Rentiere “starten”, santa ausführen

```
main :: IO ()
main = do buf ← setupBufferListener

          elfGroup ← newGroup 3
          sequence_ [ elf buf elfGroup id |
                      id ← [1 .. 10] ]
          deerGroup ← newGroup 9
          sequence_ [ reindeer buf deerGroup id |
                      id ← [1 .. 9]]
          forever (santa buf elfGroup deerGroup)
```

Zusammenfassung

- *The future is now, the future is concurrent*
- Lock-basierte Nebenläufigkeitsansätze skalieren schlecht
  - Korrekte Einzelteile können nicht ohne weiteres komponiert werden
- Software Transactional Memory als Lock-freie Alternative
  - Atomarität (atomically), Blockieren (retry), Choice (orElse) als Fundamente kompositionaler Nebenläufigkeit
  - Faszinierend einfache Implementierungen gängiger Nebenläufigkeitsaufgaben
- Das freut auch den Weihnachtsmann:
  - Santa Claus Problem in STM Haskell

Literatur

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy.
Composable memory transactions.
In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

Simon Peyton Jones.
Beautiful concurrency.
In Greg Wilson, editor, *Beautiful code*. O'Reilly, 2007.

Herb Sutter.
The free lunch is over: a fundamental turn toward concurrency in software.
Dr. Dobb's Journal, 30(3), March 2005.