

Reaktive Programmierung  
Vorlesung 13 vom 19.06.19  
Software Transactional Memory

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2019

# Fahrplan

- ▶ Einführung
- ▶ Monaden und Monadentransformer
- ▶ Nebenläufigkeit: Futures and Promises
- ▶ Aktoren I: Grundlagen
- ▶ Aktoren II: Implementation
- ▶ Meta-Programmierung
- ▶ Bidirektionale Programmierung
- ▶ Reaktive Ströme I
- ▶ Reaktive Ströme II
- ▶ Funktional-Reaktive Programmierung
- ▶ Software Transactional Memory
- ▶ Eventual Consistency
- ▶ Robustheit und Entwurfsmuster
- ▶ Theorie der Nebenläufigkeit, Abschluss

# Heute gibt es:

- ▶ Motivation: Nebenläufigkeit tut not!

# Heute gibt es:

- ▶ Motivation: Nebenläufigkeit tut not!
- ▶ Einen fundamental 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
  - ▶ Weihnachtlich: das Santa Claus Problem

# Aktueller Stand der Technik

- ▶ C: Locks und conditional variables

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

- ▶ Java (Scala): Monitore

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

- ▶ Haskell: MVars

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

# Stand der Technik: Locks und Conditional variables

- ▶ 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)

# Stand der Technik: Locks und Conditional variables

- ▶ 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**

# Stand der Technik: Locks und Conditional variables

- ▶ 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: **Lock-basierte Programme sind nicht komponierbar!**
  - ▶ Korrekte Einzelbausteine können zu fehlerhaften Programmen zusammengesetzt werden
- ▶ Klassisches Beispiel: Übertragung eines Eintrags von einer Map in eine andere
  - ▶ Map-Bücherei explizit thread-safe, d.h. nebenläufiger Zugriff sicher
  - ▶ Implementierung der Übertragung:

```
transferItem item c1 c2 = do
  delete c1 item
  insert c2 item
```

- ▶ Problem: Zwischenzustand, in dem item in keiner Map ist
- ▶ Plötzlich doch wieder Locks erforderlich! Welche?

## Kritik am Lock-basierten Ansatz (3)

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

```
void foo(void) {  
...  
    select(k1, r1, w1, e1, &t1);  
...  
}
```

```
void bar(void) {  
...  
    select(k2, r2, w2, e2, &t2);  
...  
}
```

- ▶ **Keine** Möglichkeit, `foo()` und `bar()` zu komponieren, so dass bspw. auf `r1` und `r2` gewartet wird

# STM: software transactional memory

## Grundidee: Drei Eigenschaften

- 1 Transaktionen sind **atomar**
- 2 Transaktionen sind **bedingt**
- 3 Transaktionen sind **komponierbar**

- ▶ Eigenschaften entsprechen Operationen:
  - ▶ Atomare Transaktion
  - ▶ Bedingte Transaktion
  - ▶ Komposition von Transaktionen
- ▶ Typ STM von Transaktionen (Monad)
- ▶ Typsystem stellt sicher, dass Transaktionen reversibel sind

# Transaktionen sind atomar

- ▶ 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 }
```

# 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 blockiert, wenn keine Items verfügbar



# Transaktionen sind bedingt

- ▶ Kompositionales “Blockieren” mit `retry`
- ▶ Idee: ist notwendige Bedingung innerhalb einer Transaktion 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 (worauf?)

# Transaktionen sind bedingt

- ▶ Kompositionales “Blockieren” mit `retry`
- ▶ Idee: ist notwendige Bedingung innerhalb einer Transaktion 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 Änderung an in Transaktion **gelesenen** Variablen!
  - ▶ Genial: System verantwortlich für Verwaltung der Aufweckbedingung
- ▶ Keine lost wakeups, keine händische Verwaltung von conditional variables

# Transaktionen sind kompositional

- ▶ 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:
  - ▶ Bsp: `atomically $ do { if (done) delete_file (important); S2 }`
  - ▶ Idee: Seiteneffekte werden auf **Transaktionsspeicher** beschränkt
- ▶ Ideal: Trennung wird **statisch** erzwungen
  - ▶ In Haskell: Trennung im **Typsystem**
  - ▶ IO-Aktionen vs. STM-Aktionen (Monaden)
  - ▶ Innerhalb der STM-Monade nur **reine** Berechnungen (kein IO!)
  - ▶ STM Monade erlaubt **Transaktionsreferenzen** TVar (ähnlich IORef)

# Software Transactional Memory in Haskell

- ▶ Kompakte Schnittstelle:

```
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 ()
```

- ▶ Passt auf eine Folie!

# 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 zugriffene Speicherstelle hat zum Prüfzeitpunkt denselben Wert wie beim **ersten** Lesen

## 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 []  
        return $ B tv
```

- ▶ 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  
                writeTVar tv (xs ++ [x])
```



## Puffer mit STM: Modul MyBuffer (2)

- ▶ 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'
                          return y
```

## Puffer mit STM: Anwendungsbeispiel

```
useBuffer :: IO ()
useBuffer = do
  b ← atomically $ new
  forkIO $ forever $ do
    n ← randomRIO(1,5)
    threadDelay (n*106)
    t ← getCurrentTime
    mapM_ (λx → atomically $ put b $ show x) (replicate n t)
  forever $ do x ← atomically $ get b
               putStrLn $ x
```

# Anwendungsbeispiel Philosophers.hs

- ▶ Gesetzlich vorgeschrieben als Beispiel
- ▶ Gabel als TVar mit Zustand Down oder Taken, und einer Id:

```
data FS = Down | Taken deriving Eq  
data Fork = Fork { fid :: Int, tvar :: TVar FS }
```

- ▶ Am Anfang liegt die Gabel auf dem Tisch:

```
newFork :: Int → IO Fork  
newFork i = atomically $ do  
  f ← newTVar Down  
  return $ Fork i f
```

Uses code from  
[http://rosettacode.org/wiki/Dining\\_philosophers#Haskell](http://rosettacode.org/wiki/Dining_philosophers#Haskell)

# Anwendungsbeispiel Philosophers.hs

- ▶ Transaktionen:
- ▶ Gabel aufnehmen— kann fehlschlagen

```
takeFork :: Fork → STM ()  
takeFork (Fork _ f) = do  
  s ← readTVar f  
  when (s == Taken) retry  
  writeTVar f Taken
```

- ▶ Gabel ablegen— gelingt immer

```
releaseFork :: Fork → STM ()  
releaseFork (Fork _ f) = writeTVar f Down
```

# Anwendungsbeispiel Philosophers.hs

- ▶ Ein Philosoph bei der Arbeit (putStrLn elidiert):

```
runPhilosopher :: String → (Fork, Fork) → IO ()
runPhilosopher name (left, right) = forever $ do
  delay ← randomRIO (1, 50)
  threadDelay (delay * 100000) — 1 to 5 seconds
  atomically $ do {takeFork left; takeFork right}
  delay ← randomRIO (1, 50)
  threadDelay (delay * 100000) — 1 to 5 seconds.
  atomically $ do {releaseFork left; releaseFork right}
```

- ▶ Atomare Transaktionen: beide Gabeln aufnehmen, beide Gabeln ablegen

# Santa Claus Problem

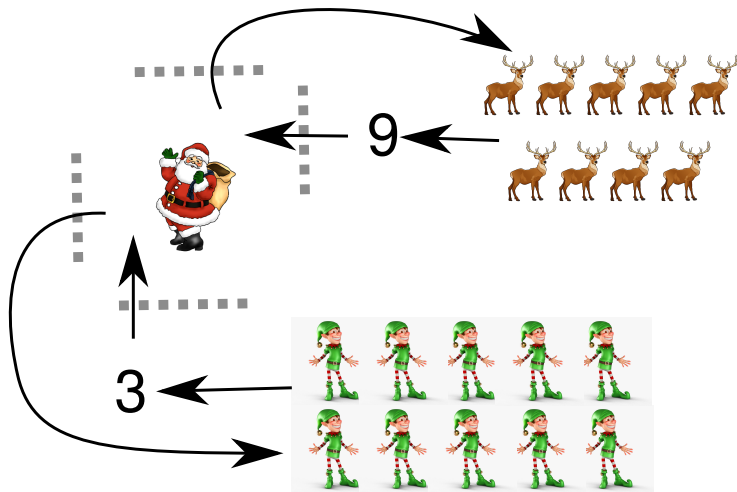
Ein modernes Nebenläufigkeitsproblem:

*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.*

aus:

J. A. Trono, *A new exercise in concurrency*, SIGCSE Bulletin, 26:8–10, 1994.

# Santa Claus Problem, veranschaulicht



# Lösungsstrategie

- ▶ Modellieren jede Elfe, jedes Rentier, und den 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, und den 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, und den 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)
```



# Eine Gruppe erwarten

- ▶ Santa erwartet Elfen und Rentiere in entsprechender Gruppengröße
- ▶ Erzeugt neue Gatter für nächsten Rutsch
  - ▶ Verhindert, dass Elfen/Rentiere sich “hineinmogeln”

```
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
  - ▶ **A**tomarität ( *atomically* ), **B**lockieren ( *retry* ), **C**hoice ( *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. Dobbs' Journal*, 30(3), March 2005.