GOETHE UNIVERSITÄT FRANKFURT AM MAIN

www.uni-frankfurt.de

Lock-freies nebenläufiges Programmieren durch Software Transactional Memory

David Sabel

Goethe-Universität, Frankfurt am Main

10. Juni 2013

Stand der Folien:

Motivation

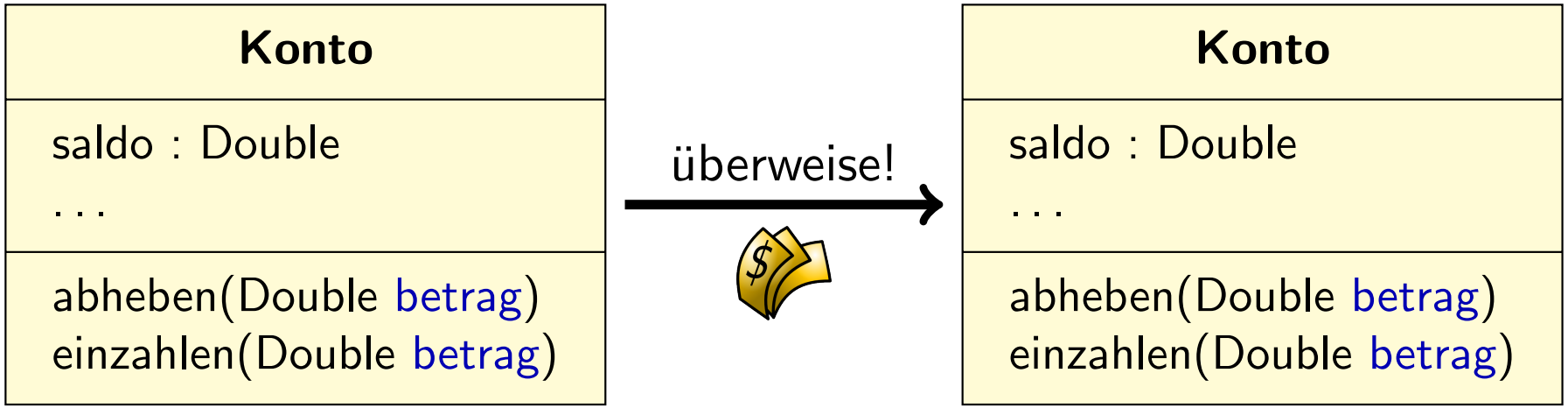
Parallele, **nebenläufige** und **verteilte** Programme gewinnen an **Bedeutung**:

- Zunehmende **Anzahl an Prozessoren**
- Zunehmende **Vernetzung** von Rechnern

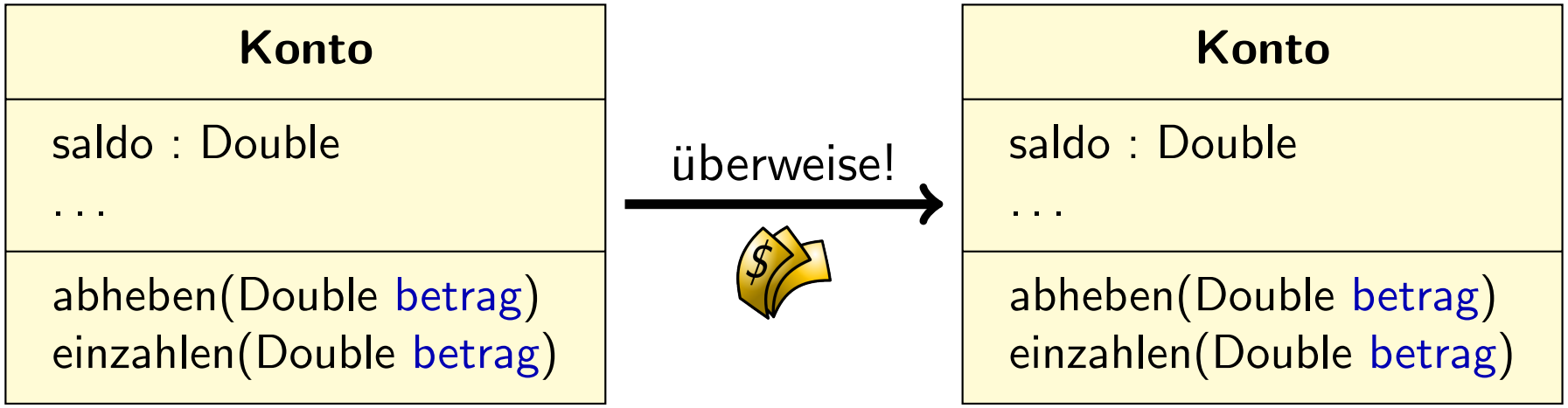
Aber:

- Programmierung muss **Synchronisation** von Prozessen beachten
- Dies ist oft **schwierig** und **fehleranfällig**
- **Abhilfe**: Neue Programmieransätze und Bibliotheken

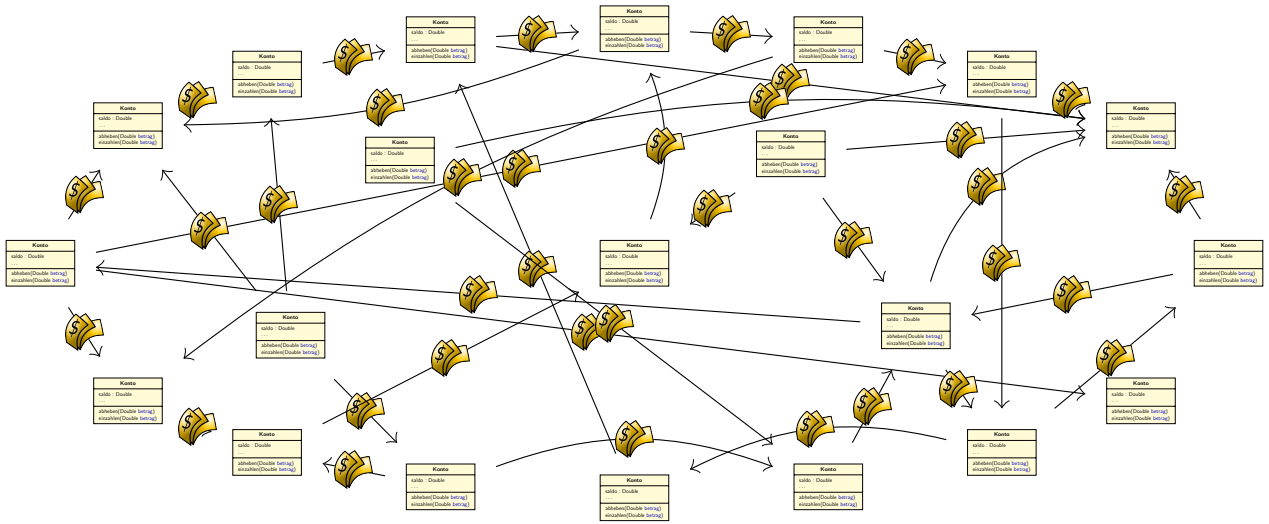
Beispiel: Lock-basierte Programmierung



Beispiel: Lock-basierte Programmierung



im **nebenläufigen** System: konkurrierende Threads (Überweisungen)



Beispiel: Lock-basierte Programmierung

Konto
saldo : Double ...
lock() unlock() abheben(Double betrag) einzahlen(Double betrag)

Implementierung von abheben / einzahlen:

```
abheben(Double betrag) {
  lock();
  saldo := saldo - betrag;
  unlock()}

einzahlen(Double betrag) {
  abheben(-betrag)}
```

Aufgabe

Implementiere ueberweise(Konto von, Konto nach, Double betrag) **atomar**:

Zwischenzustände dürfen **nicht beobachtbar** sein

„Summe aller Salden bleibt stets gleich“

Beispiel: Lock-basierte Programmierung

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Konto
saldo : Double ...
lock() unlock() abheben(Double betrag) einzahlen(Double betrag)

Implementierung von abheben / einzahlen:

```
abheben(Double betrag) {
  lock();
  saldo := saldo - betrag;
  unlock()}

einzahlen(Double betrag) {
  abheben(-betrag)}
```

Überweisen: 1. Versuch

```
ueberweise(Konto von, Konto nach, Double betrag) {
        abheben(von,betrag);
        einzahlen(nach,betrag)
}
```

Beispiel: Lock-basierte Programmierung

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Konto
saldo : Double ...
lock() unlock() abheben(Double betrag) einzahlen(Double betrag)

Implementierung von abheben / einzahlen:

```
abheben(Double betrag) {
  lock();
  saldo := saldo - betrag;
  unlock()}

einzahlen(Double betrag) {
  abheben(-betrag)}
```

Überweisen: 1. Versuch

```
ueberweise(Konto von, Konto nach, Double betrag) {
        abheben(von,betrag);
        einzahlen(nach,betrag)
}
```

← **Zwischenzustand beobachtbar**

Erfüllt nicht die Anforderung!

Beispiel: Lock-basierte Programmierung

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Konto
saldo : Double ...
lock() unlock() abheben(Double betrag) einzahlen(Double betrag)

Implementierung von abheben / einzahlen:

```
abheben(Double betrag) {
  lock();
  saldo := saldo - betrag;
  unlock()}

einzahlen(Double betrag) {
  abheben(-betrag)}
```

Überweisen: 2. Versuch

```
ueberweise(Konto von, Konto nach, Double betrag) {
        von.lock();nach.lock();
        abheben(von,betrag);
        einzahlen(nach,betrag);
        von.unlock();nach.unlock() }
```

Beispiel: Lock-basierte Programmierung

Konto
saldo : Double ...
lock() unlock() abheben(Double betrag) einzahlen(Double betrag)

Implementierung von abheben / einzahlen:

```
abheben(Double betrag) {
  lock();
  saldo := saldo - betrag;
  unlock()}

einzahlen(Double betrag) {
  abheben(-betrag)}
```

Überweisen: 2. Versuch

```
ueberweise(Konto von, Konto nach, Double betrag) {
        von.lock();nach.lock();
        abheben(von,betrag);
        einzahlen(nach,betrag);
        von.unlock();nach.unlock() }
```

Deadlock!

Beispiel: Lock-basierte Programmierung

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

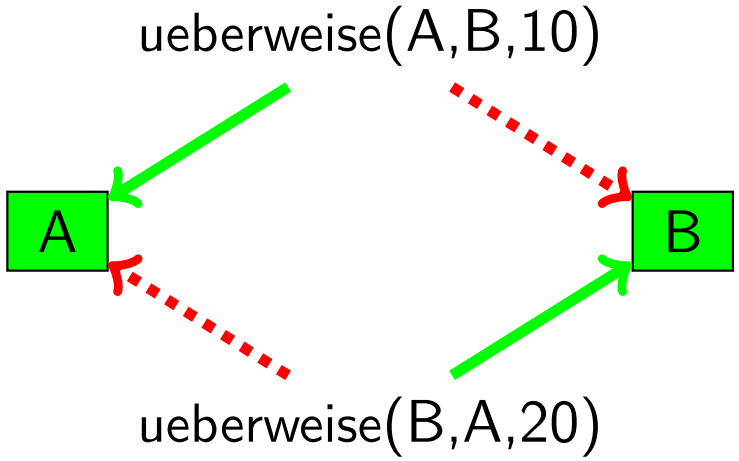
Konto
saldo : Double ...
lock() unlock() abheben(Double betrag) einzahlen(Double betrag)

Überweisen: 3. Versuch

```
ueberweise(Konto von, Konto nach, Double betrag) {
  von.lock();nach.lock();
  von.saldo := von.saldo - betrag;
  nach.saldo := nach.saldo + betrag;
  von.unlock();nach.unlock() }
```

Beispiel: Lock-basierte Programmierung

Konto
saldo : Double ...
lock() unlock() abheben(Double betrag) einzahlen(Double betrag)



Überweisen: 3. Versuch

```
ueberweise(Konto von, Konto nach, Double betrag) {
  von.lock();nach.lock();
  von.saldo := von.saldo - betrag;
  nach.saldo := nach.saldo + betrag;
  von.unlock();nach.unlock() }
```

Deadlock!

Beispiel: Lock-basierte Programmierung

Konto
saldo : Double ...
lock() unlock() abheben(Double betrag) einzahlen(Double betrag)

Funktioniert, aber jeder muss
totale Ordnung kennen und einhalten

Überweisen: 4. Versuch

```
ueberweise(Konto von, Konto nach, Double betrag) {
  if von < nach then {von.lock();nach.lock()} else {nach.lock;von.lock};
  von.saldo := von.saldo - betrag;
  nach.saldo := nach.saldo + betrag;
  von.unlock();nach.unlock() }
```

Varianten des Beispiels

- **Warte**, wenn Konto nicht gedeckt ist:
  Standardlösung: Condition Variables
  ➳ Sogar der Code von einzahlen muss angepasst werden
- Wenn Konto A nicht gedeckt, überweise von Konto B
  ➳ Lässt sich nicht aus bestehendem Code komponieren
  ➳ Komplett neu programmieren
- Wenn A.saldo + B.saldo $\geq$ betrag, dann überweise auf C, sonst warte
  ➳ Verwaltung mit Condition Variablen noch möglich?

Probleme der Lock-basierte Programmierung

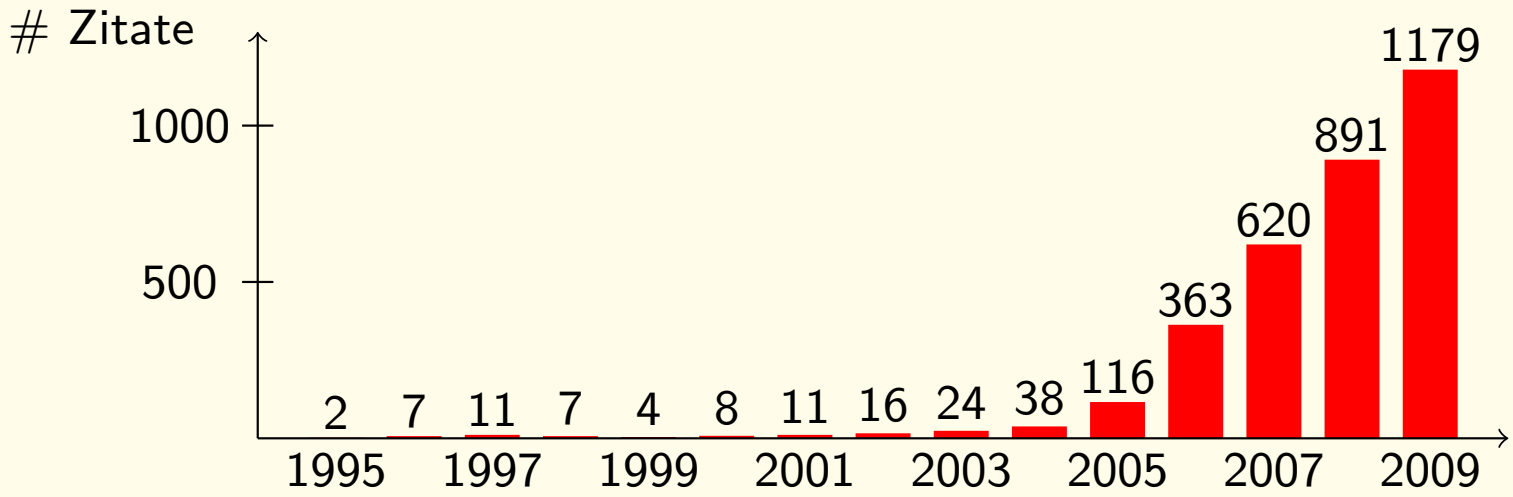
(vgl. Peyton Jones: Beautiful concurrency, O'Reilly, 2007)

- Setzen von **zu wenig** Locks: Vergessenes Sperren führt zu unkontrolliertem Verhalten (Race Conditions)
  „Geldmenge hängt von der Ausführung ab"
- Setzen von **zu vielen** Locks:
  - Programm unnötig sequentiell (Best-Case)
    „Ein globaler Lock $\Longrightarrow$ *Nur eine Überweisung zu einer Zeit"*
  - Deadlock (Worst-Case)
- Setzen der **falschen** Locks: Beziehung zwischen Sperren und Daten nicht immer offensichtlich
- Lock-basierte Programmierung ist nicht **modular**!

Transactional Memory

- Programmieransatz **ohne Locks**
- Anwender programmiert **Transaktionen**, die analog zu Datenbankentransaktionen ausgeführt werden:
  - Transaktionsmanager überwacht Ausführung der Transaktionen
  - Logging der Einzeloperationen
  - Rollback & Restart im Konfliktfall
- **Transaktionen** auf dem **gemeinsamen Speicher**
- **System** garantiert **Deadlockfreiheit**
- **Software Transactional Memory** (STM): Implementierung als Programm-Bibliothek

Software Transactional Memory: Einige Arbeiten



Microsoft Academic Search, Keyword „Software Transactional Memory“

- Shavit & Touitou PODC'95: Software transactional memory (erster Artikel)
- Harris et. al. PPoPP'05: Composable Memory Transactions (STM Haskell, orElse)
- Cascaval et. al. CACM 08: STM: why is it only a research toy? (Kritik)
- Bücher zu Transactional Memory:
  - Larus & Rajwar 2007: Transactional Memory
  - Guerraoui & Kapalka 2010: Principles of Transactional Memory

STM: Primitive

Atomare Blöcke: **atomic** { Transaktionaler Code }

- kennzeichnet **Codeblock** als **eine Transaktion**
- Ablauf: **Ganz oder gar nicht**,
  Roll-back und Restart bei Konflikt (nicht beobachtbar)

Beispiel: ueberweise(von,nach,betrag)

```
atomic {
    von.abheben(betrag);
    nach.einzahlen(betrag);
}
```

Transaktionale Variablen: Variablen, die innerhalb von Transaktionen gelesen / geschrieben werden

STM: Primitive (2)

Koordination von Transaktionen: retry

- Transaktion wird **abgebrochen** (Roll-Back) und **erneut gestartet**
- Implementierungen: **Warten**, bis sich Speicherzustand ändert

Beispiel: Überweisen, nur wenn Konto gedeckt:

```
ueberwBedingt(von,nach,betrag) {
  atomic {
     if von.saldo ≥ betrag then
           { ueberweise(von,nach,betrag) }
     else {retry}
  }
}
```

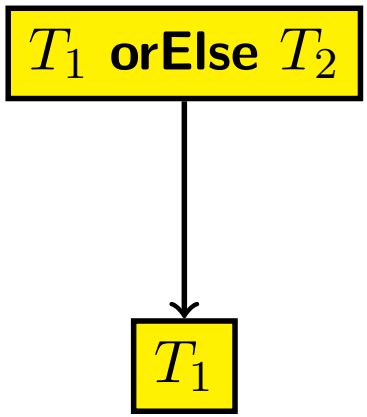
STM: Primitive (3)

Alternativoperator: Transaktion1 **orElse** Transaktion2

T_1 **orElse** T_2

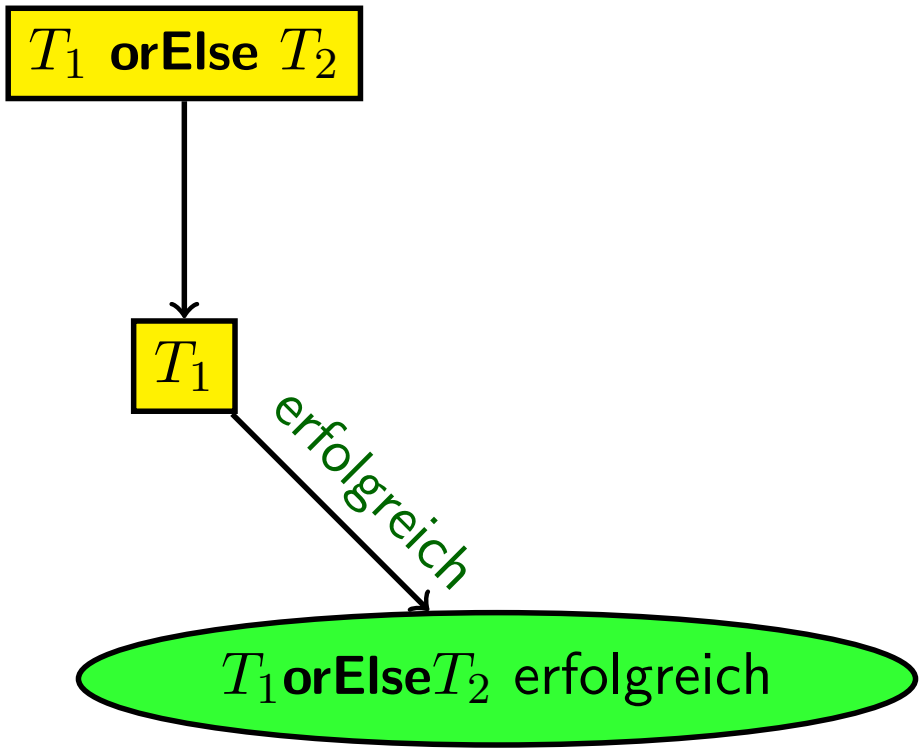
STM: Primitive (3)

Alternativoperator: Transaktion1 **orElse** Transaktion2



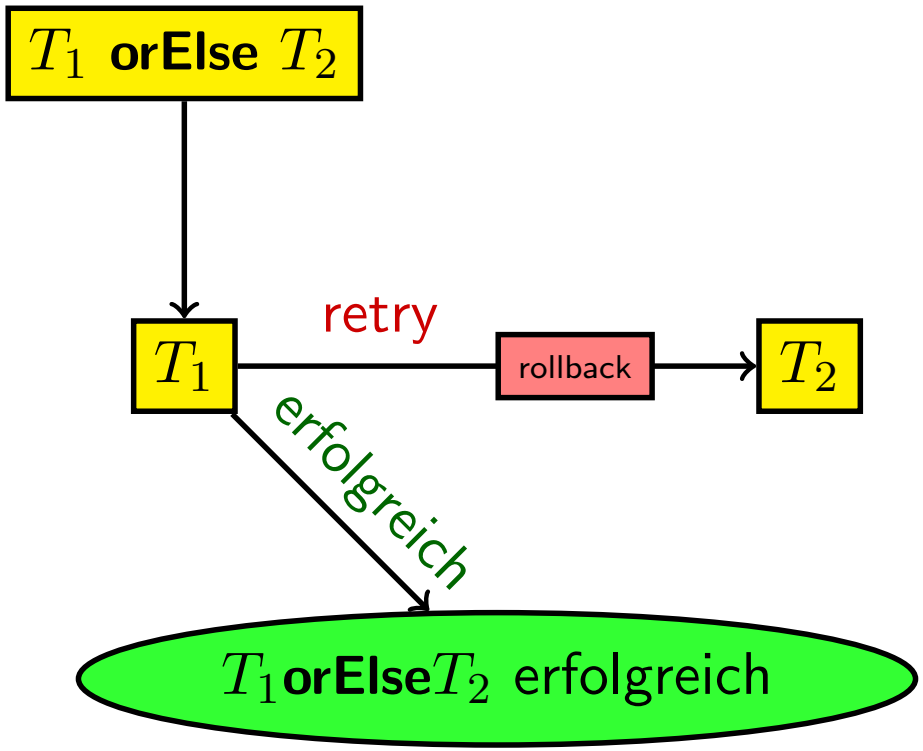
STM: Primitive (3)

Alternativoperator: Transaktion1 **orElse** Transaktion2



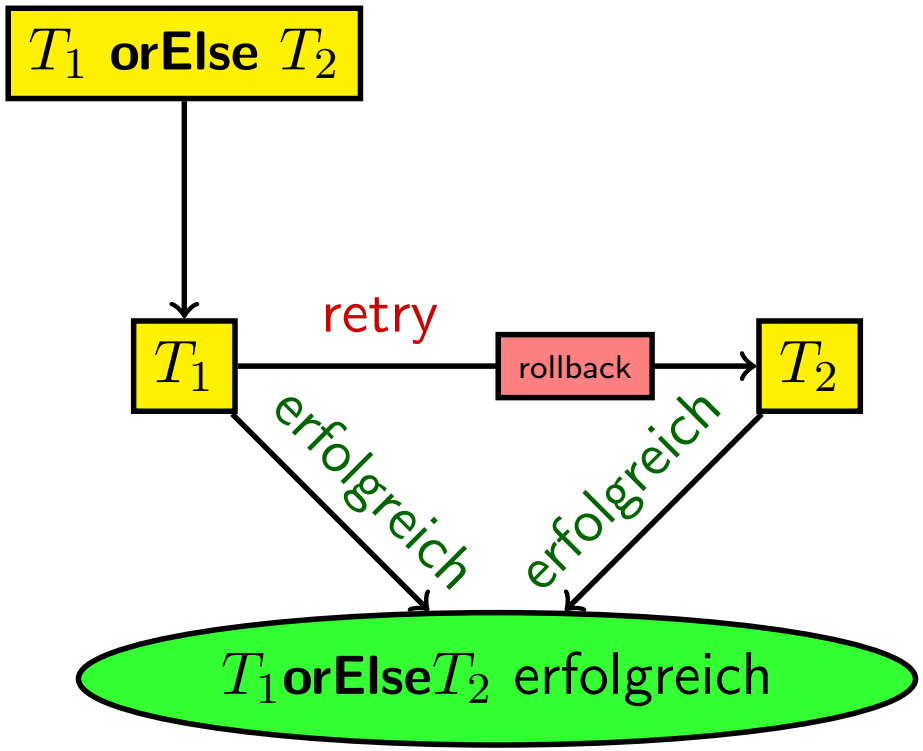
STM: Primitive (3)

Alternativoperator: Transaktion1 **orElse** Transaktion2



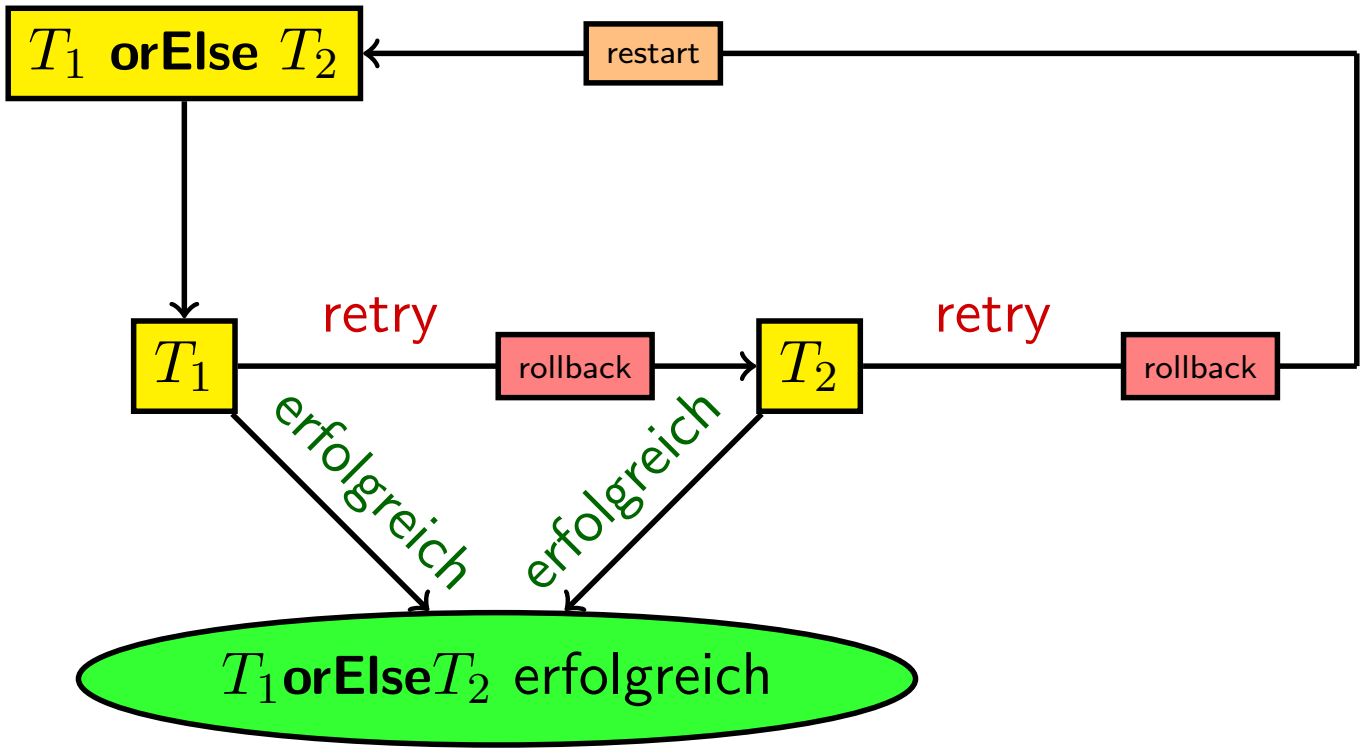
STM: Primitive (3)

Alternativoperator: Transaktion1 **orElse** Transaktion2



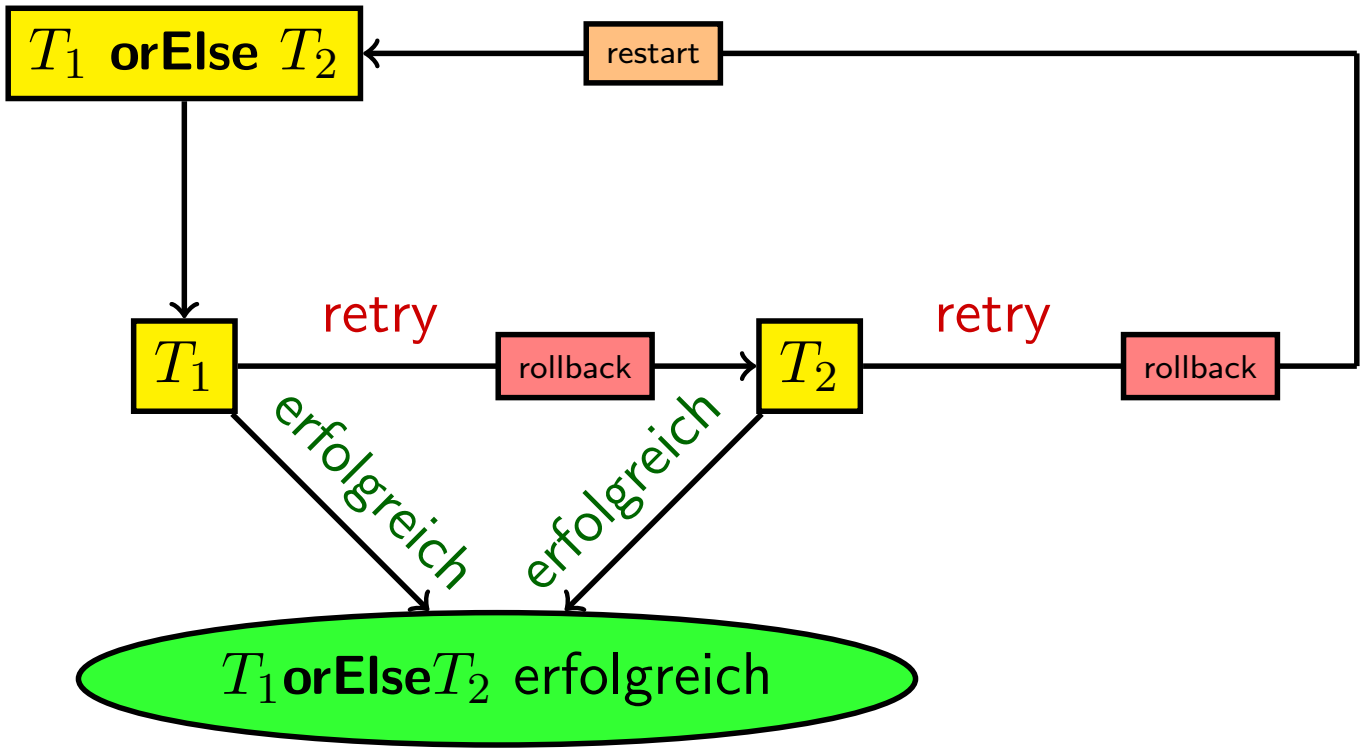
STM: Primitive (3)

Alternativoperator: Transaktion1 **orElse** Transaktion2



STM: Primitive (3)

Alternativoperator: Transaktion1 **orElse** Transaktion2



Beispiel: Alternatives Überweisen:

atomic {ueberwBedingt(A,C,betrag) **orElse** ueberwBedingt(B,C,betrag)}

STM vs. Datenbanktransaktionen (1)

ACID-Eigenschaften

- **Atomic**: Transaktion wird **ganz oder gar nicht** ausgeführt.

  Aber: Bei STM sind Endlosschleifen möglich / erlaubt!

  D.h. mögliche Ausgänge von STM-Transaktionen:
  erfolgreich / abgebrochen / **nicht terminierend**

- **Consistency**

- **Isolation**: Nebenläufige Ausführung nicht beobachtbar,

  Beobachtung von Zwischenwerten nicht möglich

  Effekt wie sequentielle Hintereinanderausführung

- **Durability**: Gilt nicht für STM, da flüchtiger Speicher

STM vs. Datenbanktransaktionen (2)

Weitere Unterschiede:

- STM muss in bestehende Programmiersprachen **integriert** werden, **keine unabhängige Umgebung**
- Datenbanken können Zugriffszeiten auf Festplatten mit Rechenzeit **gegenrechnen**, im Hauptspeicher geht das nicht, da **Zugriffszeiten viel kürzer**

Integration in Programmiersprachen: Probleme

- Atomarität / Isolation erfordert:

  Alle Zugriffe auf transaktionale Variablen **durch Transaktionen**

- **Geschachtelte** Transaktionen:

  atomic { . . . **atomic** { . . . } . . . }

  Semantik unklar, am besten **verbieten**, wie?

- Transaktionen müssen **umkehrbar** sein:

Integration in Programmiersprachen: Probleme

- Atomarität / Isolation erfordert:

  Alle Zugriffe auf transaktionale Variablen **durch Transaktionen**

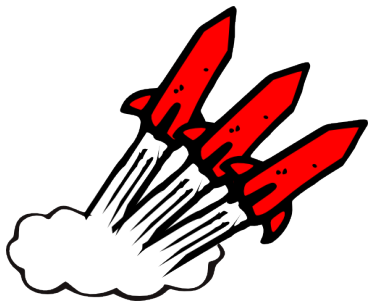
- **Geschachtelte** Transaktionen:

  atomic { . . . **atomic** { . . . } . . . }

  Semantik unklar, am besten **verbieten**, wie?

- Transaktionen müssen **umkehrbar** sein:

```
atomic {
    launchMissiles;
    retry }
```



Integration in Haskell

Haskell allgemein:

- **Rein-funktionale** Programmiersprache
- Starkes, statisches **Typsystem**
- Seiteneffekte: **Nur in der `IO`-Monade** erlaubt

```
return  :: a -> IO a
(>>=)   :: IO a -> (a -> IO b) -> IO b
putChar :: Char -> IO ()
getChar :: IO Char
```

  Verboten: `unIO :: IO a -> a` **!**

- Monade sichert sequentielle Abarbeitung zu
- Typsystem **unterscheidet** Seiteneffekt-basierte Programmierung von funktionaler Programmierung
- Concurrent Haskell: `forkIO :: IO a -> IO ThreadId`

STM Haskell (Harris et.al, PPoPP'05)

Transaktionen **gekapselt** durch eigene `STM`-Monade

- `return :: a -> STM a`
- `(>>=)  :: STM a -> (a -> STM b) -> STM b`
- `retry  :: STM ()`
- `orElse :: STM a -> STM a -> STM a`

Transaktionale Variablen: `TVar a`

- `newTVar   :: a -> STM (TVar a)`
- `readTVar  :: TVar a -> STM a`
- `writeTVar :: TVar a -> a -> STM ()`

STM Haskell (2)


```
atomically :: STM a -> IO a
```

führt STM-Transaktion aus

Es gibt **keine Funktion** vom Typ `IO a -> STM a`

- Zugriff auf TVars **nur innerhalb von Transaktionen** möglich
- **Geschachtelte Transaktionen** durch Typisierung **verboten**
- Keine `IO`-Aktionen innerhalb von Transaktionen
  dadurch: **Umkehrbarkeit garantiert!**

Korrektheit von STM-Implementierungen

- Es gibt viele **Korrektheitsbegriffe**, z.B.

  Isoliertheit, Linearisierbarkeit, Sequentialisierbarkeit, Wiederherstellbarkeit, Opakheit, Atomarität, globale Atomarität, (starke) Fortschrittschlichkeit, ...

  Übersicht: siehe z.B. Guerraoui & Kapalka, PPoPP'08

- Die meisten Begriffe beschreiben

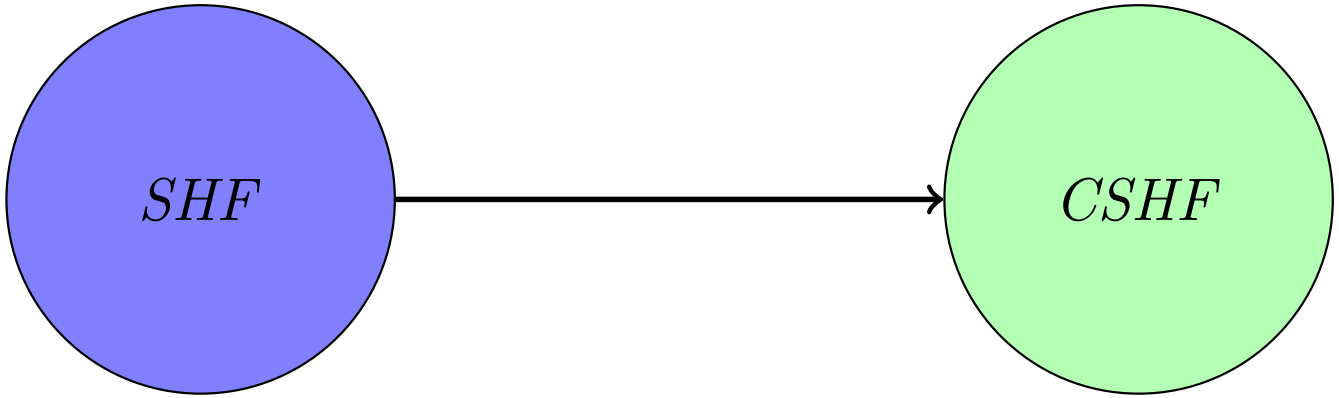
  Eigenschaften des erzeugten Trace

  der Lese-/Schreibzugriffe auf transaktionale Variablen

- **Semantischer** Ansatz: Schmidt-Schauß & S., ICFP'13, akzept.

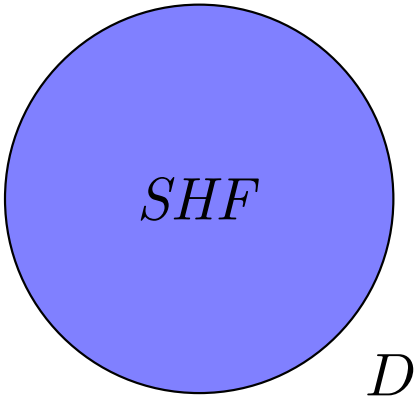
SSS'13: Ansatz

STM Haskell: Eine Sprache **zwei** Semantiken



Spezifikation Implementierung

SSS'13: Spezifikation: *SHF*



- Operationale Semantik $\xrightarrow{SHF}$, die Transaktionen **sequentiell hintereinander** ausführt
- Z.B. Big-Step-Regel für `atomically`:

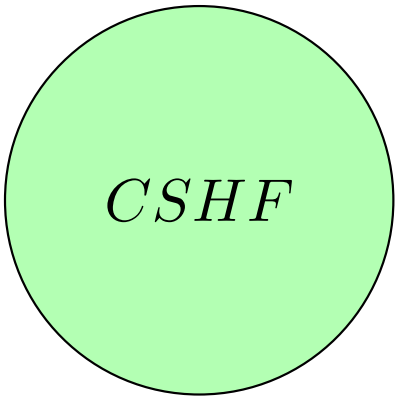
$$\frac{D[\texttt{atomically}\ e] \xrightarrow{SHFA,*} D'[\texttt{atomically}\ (\texttt{return}\ v)]}{D[\texttt{atomically}\ e] \xrightarrow{SHF} D'[\texttt{return}\ v]}$$

- Offensichtlich **korrekt**: Transaktionen stören sich nicht
- **Keinerlei Nebenläufigkeit** der Transaktionen!
- **Keine** Implementierung, denn:

  Die **Anwendbarkeit** obiger Regel ist **unentscheidbar**

  (Terminierung von $\xrightarrow{SHFA}$ ist nicht entscheidbar)

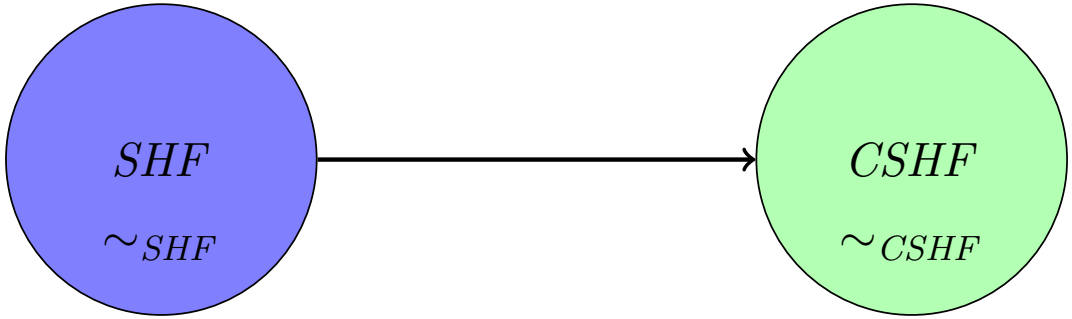
SSS'13: Implementierung: *CSHF*



- Operationale Semantik $\xrightarrow{CSHF}$, die Transaktionen **stückchenweise** und **nebenläufig** ausführt.
- Lokales Transaktions-Log (**pessimistisch**)
- **Konflikterkennung**: Während des Commits werden in Konflikt geratene Transaktionen **benachrichtigt** und **direkt abgebrochen**

- Regelanwendungen **entscheidbar** und **sehr nebenläufig**
- Haskell-Prototyp verfügbar
- **Korrektheit** nicht offensichtlich

SSS'13: Korrektheit der Implementierung



Theorem

Konvergenzäquivalenz: Für jeden SHF-Prozess P gilt:

$$P \downarrow_{SHF} \iff P \downarrow_{CSHF} \text{ und } P \Downarrow_{SHF} \iff P \Downarrow_{CSHF}$$

Adäquatheit: Für alle $P_1, P_2 \in SHF$ gilt:

$$P_1 \sim_{CSHF} P_2 \implies P_1 \sim_{SHF} P_2$$

- $CSHF$ ist ein **korrekter Auswerter** für SHF
- Korrekte Transformationen in $CSHF$ sind korrekt für SHF

Fazit

- STM **behebt viele Probleme** der Lock-basierten Programmierung
- Verwendung von STM ist **sehr einfach**
- Implementierung und Korrektheit **schwierig**, aber **möglich**
- STM noch zu **langsam**, daher **weitere Forschung notwendig**
- **Nicht alle** Probleme sind lösbar mit STM: Umkehrbarkeit der Transaktionen notwendig