14. Programme unter schwachen Speichermodellen

Ziele: (1) Semantik von Programmen unter schwachen Speichermodellen verstehen

(2) Verstehen wie schwache Speichermodelle funktionieren
↳ Schwache Speichermodelle formalisieren in einem uniformen, axiomatischen Framework (CAT)

(3) Verifikation von Programmen unter schwachen Speichermodellen

Dazu:
- Wir wollen das Konzept der "Happens-before-traces" von TSO verallgemeinern.
- Wir wollen keine SOS Semantik für Speichermodelle definieren, sondern axiomatisch (!) arbeiten.

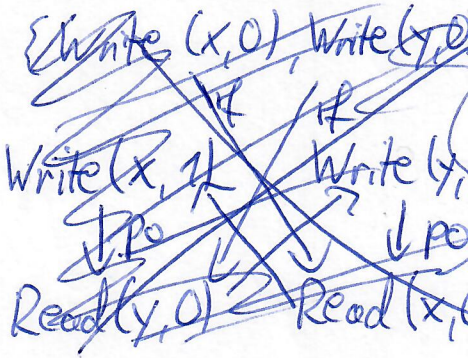
Bem: Die folgenden Inhalte sind an Jade Alglave's Arbeiten angelehnt, stimmen aber nicht mit diesen überein!

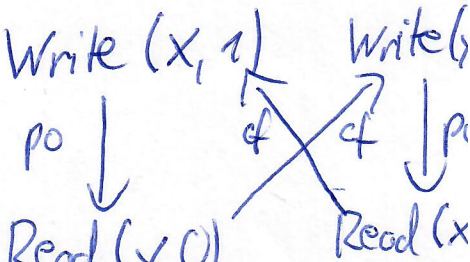
VON

```
int g_x = 0;
int g_y = 0;

int e;                 ||  int e';
g_x = 1;               ||  g_y = 1;
e = g_y;               ||  e' = g_x;
if (e == 0) then       ||  if (e' == 0) then
    critical section 1;||      critical section 2;
fi                     ||  fi
```

ZU





14.1 Syntax und "anarchische" Semantik von Programmen

Ziel:
- Definiere anarchische Semantik von Programmen
  ↳ Unabhängig von jeglichen Speichermodellen !

Bisher:
- SOS für SC: Atomarer Speicher + Interleaving
- SOS für TSO: Hauptspeicher + Buffer + Interleaving
- SOS für PowerPC: Speicher pro Thread + Store/Load-Buffer + Interleaving + Speculation + ???

Probleme:
- Semantische Domäne der Programme hängt vom Speichermodell ab.
- SOS definiert semantische Domäne (z.B. Hauptspeicher + Store-Buffer/Thread) UND Transformationsregeln (EARLY, ISSUE, STORE, ...)
- Viele "Implementierungsdetails"
  ↳ TSO kann auch über Load-Buffer implementiert werden (Dual TSO, Paresh Aziz Abdulla CONCUR16
- Ohne Speichermodell hat Programm Keine Semantik

Lösungsansätze:
- Definiere Grundsemantik von Programmen (SOS), die (nahezu) alles erlaubt.
- Speichermodelle definieren Restriktionen auf dieser Grundsemantik.
  ↳ Ohne Speichermodell => Maximale Semantik

- Programm ohne Speichermodell hat auch keinen Heap/Speicher. Trotzdem gibt es Heap/Speicher-Operationen. Wie?
  - ↳ Behalte Stack bei
  - ↳ Write-Operationen tun nichts
  - ↳ Read-Operationen lesen beliebigen Wert
  - ↳ Alloc/Dealloc erlauben wir nicht (zur Einfachheit).
  - ↳ Lokale Operationen funktionieren wie erwartet

Zusätzlich erlauben wir keine Pointerarithmetik:

- Jeder Thread hat lokale Registervariablen (R1, R2, ...)
- Es gibt globale Variablen (x, y, z, ...)

Formaler: Wir adaptieren die Syntax von W--:

- Ausdrücke "a" dürfen keine globalen Variablen enthalten
- Wir haben 3 Arten von Zuweisungen:
  (1) x := a (Write auf x im Speicher)
  (2) R := x (Read von x aus Speicher)
  (3) R := a (Lokale Zuweisung)

Zusätzlich sollen Ausführungen eines Thread neben einer Sequenz lokaler Konfigurationen auch eine Sequenz folgender Events erzeugen

- Read(x, val) $x \in GVar(P)$
- Write(x, val) $val \in \mathbb{Z}$
- Branch(b) $b \in \{0, 1\}$
- Local
- (potenziell weitere wie z.B. Fences)

Neben den Parametern der Events, gehen wir davon aus, dass jedem Event e

- ein Thread $thr(e)$,
- die erzeugende Instruktion $instr(e)$ und
- ein lokaler Timestamp $\Theta(e) \in Time(thr(e))$

zugeordnet sind.
Dabei ist $Time(t)$ eine total geordnete Menge an "Timestamps" von Thread t. Die Timestamps verschiedener Threads sind disjunkt und unvergleichbar.
Dies hat folgende Vorteile:

(1) Timestamps erlauben gleichartige Events zu unterscheiden
(2) Sie führen ein Konzept lokaler Zeit ein, ohne ein Konzept von globaler Zeit einzuführen!

Formal (SOS): Wir adaptieren die SOS Semantik von W--:

- (SKIP), (ASSUME) und (ASSIGN) produzieren das Event Local (Bsp: $(skip, \sigma) \xrightarrow{Local} \sigma$)
- (IF_TRUE) und (WHILE_TRUE) produzieren Branch(1), die Gegenstücke entsprechend Branch(0).

Zusätzlich gibt es 2 neue Regeln:

(READ) $\dfrac{}{(R:=x, \sigma) \xrightarrow{Read(x,v)} \sigma[R \leftarrow v]}$ mit $v \in \mathbb{Z}$ beliebig

(WRITE) $\dfrac{}{(x:=a, \sigma) \xrightarrow{Write(x,v)} \sigma}$ mit $v = [\![a]\!](\sigma)$

Zusätzlich gilt für alle Regeln:
- die erzeugten Events sind mit thr(e) und instr(e) versehen.
- der Timestamp $\Theta(e)$ ist beliebig.

Def (Computation):

Eine Computation eines Threads t ist eine (potenziell unendliche) Sequenz

$$\rho_t = (c_0, \sigma_0) \xrightarrow{e_1} (c_1, \sigma_1) \xrightarrow{e_2} (c_2, \sigma_2) \dots$$

mit
- c_0 = Programmcode von t
- $\sigma_0[R] = 0 \quad (\forall R \in Reg(t))$
- $\Theta(e_i) < \Theta(e_j)$, für alle $i < j$

die durch die SOS Semantik entsteht.

Eine Computation eines parallelen Programms $P = t_1 \| t_2 \| \dots \| t_k$ ist ein Tupel

$$\rho = (IW, \bigotimes_{i=1}^{k} \rho_{t_i})$$

bestehend aus
- der Menge $IW = \{Write(x,0) \mid x \in GVar(P)\}$ der initialen Writes
- dem Produkt der Thread-Computations ρ_{t_i}

Die initialen Writes haben jeder einen gesonderten Thread t_x, eine gesonderte Instruktion mit und einen beliebigen Timestamp.

Die Menge aller Computations bezeichnen wir mit Comp / Comp(P) / Comp(t). Die Menge aller Events in einer Computation ρ nennen wir Events(ρ). Diese beinhaltet IW ▽

Zur Einfachheit schreiben wir Write(p), Read(p) und Memory(p) für die Menge aller Write/~~Red~~Read/Memory Events in p.
Memory(p) = Write(p) ∪ Read(p).

Bsp. (Dekker)

```
int x;
int y;

x := 1;          ||  y := 1;
R1 := y;         ||  R2 := x;
if (R1 == 0) then ||  if (R2 == 0) then
   cs1;          ||     cs2;
fi               ||  fi
```

Eine mögliche Computation:

IW = {Write(x,0), Write(y,0)}

~~Write~~ (x:=1;..., R1→0) × (y:=1;..., R2→0)

↓ Write(x,1) ↓ Write(y,1)

(R1:=y;..., R1→0) × (R2:=x, R2→0)

↓ Read(y,0) ↓ Read(x,42)

(if... fi, R1→0) × (if... fi, R2→42)

↓ Branch(1) ↓ Branch(0)

(cs1, R1→0) × (skip, R2→42)

↓ ... ↓ Local

... R2→42

Computations behandeln bisher Kontrollfluss und Datenfluss des Programms. Kommunikation zwischen Threads wird noch nicht berücksichtigt.
Diesen Kommunikationsfluss betrachten wir jetzt.

Def (Execution):

Eine Execution eines Programms P ist ein Tupel

$$\pi = (\rho, rf)$$

wobei

- ρ eine Computation von P ist
- $rf \subseteq Write(\rho) \times Read(\rho)$ eine Read-From Relation ist,

sodass:

(Consistency) $\forall (Write(x, v_1), Read(y, v_2)) \in rf : x = y \wedge v_1 = v_2$

und

(Totality + Uniqueness) $\forall r \in Read(\rho) : \exists! w \in Write(\rho) : (w, r) \in rf$

gelten

Die Menge aller Executions bezeichnen wir mit $Exec / Exec(P)$.

Def (Anarchische Semantik):

Die anarchische Semantik eines Programms P ist definiert als $[\![P]\!] := Exec(P)$ die Menge aller seiner Executions.

Die anarchische Semantik hat keine Restriktion auf den erlaubten Kommunikationen.

Bsp (Thin-Air + Future Read):

```
P:  R1 := x;
    x := 3;
    x := R1;
```

Mögliche Exec (nur Events)

{Write(x,0)}

Read(x, 42)

↓

Write(x, 3) } rf

↓

Write(x, 42)

Bsp (Causal cycle):

```
P:  R1 := x;                 ||  R2 := y;
    if (R1 == 42) then       ||  if (R2 == 42) then
        y := 42              ||      x := 42
    fi                       ||  fi
```

IW

Exec: