14.3 Axiomatische Speichermodelle über CAT

Bisher:
- Speichermodelle sind beliebige Funktionen, die nach beliebigen Kriterien arbeiten können
- ABER: Übliche Speichermodelle arbeiten nach "simplen" Mustern.

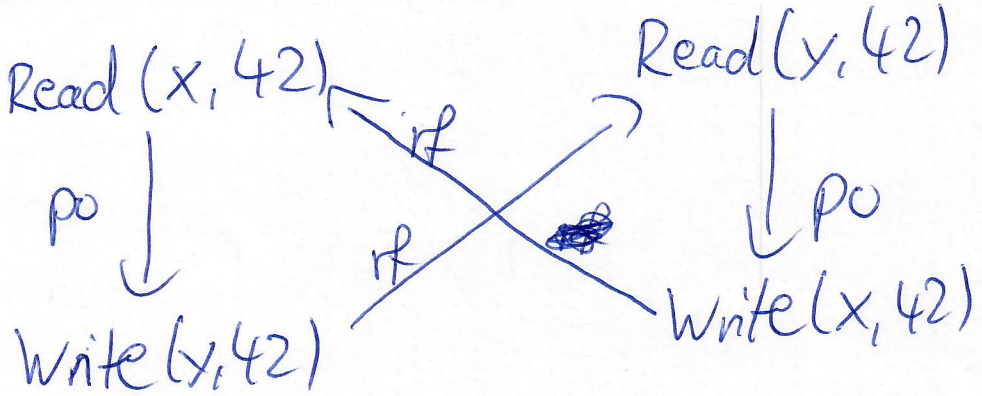
Ziel: Die CAT-Sprache zum Modellieren von axiomatischen Speichermodellen definieren.

Ansatz:
- "Ungewollte" Executions ~~erlaub~~ involvieren meist Cycles in den Abhängigkeiten
  - ↳ Speichermodelle sollen solche Cycles verbieten
  - ↳ Acyclicity ist eine graphische Eigenschaft
  - ↳ Wir wollen strukturelle ~~Eigenschaften~~ Einschränkungen formulieren

Bsp (Causal cycle):

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

Ungewollte Candidate Execution:



Strukturelle Einschränkung

$$\boxed{\text{acyclic}(po \cup rf)}$$

schließt obige Execution aus.

Constraints solcher Art werden wir in CAT ausdrücken können.

Def (Syntax von CAT):

Die Sprache CAT definiert Relationen (~ Kanten eines Graphen) und Constraints (z.B. acyclic) auf diesen.
Ein (kleiner) Teil ihrer Syntax ist gegeben durch

$$CAT ::= a \mid rel \mid CAT\ CAT$$
$$rel ::= let\ \langle name \rangle := r$$
$$r ::= base \mid \langle name \rangle \mid r_1 \cup r_2 \mid r_1 \cap r_2 \mid r_1 \setminus r_2 \mid r_1 ; r_2 \mid r_1^{-1} \mid r_1^{+} \mid r_1^{*} \mid S_1 \times S_2 \mid \dots$$
$$base ::= po \mid rf \mid loc \mid ext \mid id \mid 0 \mid \dots$$
$$S ::= \_ \mid M \mid W \mid R \mid \dots$$
$$a ::= acyclic(r) \mid irreflexive(r) \mid empty(r)$$

Ein Wort in der CAT-Sprache nennen wir CAT-File.

Def (Semantik von CAT): Gegeben eine Candidate Execution $G = (E, po, rf)$ und eine CAT-File $\mathbf{C}$, die relationelle Semantik ist wie folgt definiert:

$[\![base]\!](G) \subseteq E \times E$:

- $[\![po]\!](G) = po$, $[\![rf]\!](G) = rf$
- $[\![loc]\!](G) = \{(e_1, e_2) \mid var(e_1) = var(e_2)\}$
- $[\![ext]\!](G) = \{(e_1, e_2) \mid thr(e_1) \neq thr(e_2)\}$
- $[\![id]\!](G) = \{(e, e) \mid e \in E\}$, $[\![0]\!] = \emptyset$

...

$[\![r]\!](G) \subseteq E \times E$:

- $[\![r_1 \cup r_2]\!](G) = [\![r_1]\!](G) \cup [\![r_2]\!](G)$
- Semantik der anderen Operatoren ist wie erwartet
- $[\![S_1 \times S_2]\!](G) = [\![S_1]\!](G) \times [\![S_2]\!](G)$
- $[\![\langle name \rangle]\!](G) = lfp(X_{\langle name \rangle} = [\![r]\!](G))$,

  wobei "let $\langle name \rangle := r$" im CAT steht.

$[\![S]\!](G) \subseteq E$:

- $[\![\_]\!](G) = E$, $[\![M]\!](G) = Memory(E)$
- $[\![R]\!](G) = Read(E)$, $[\![W]\!](G) = Write(E)$

$[\![a]\!](G) \in \mathbb{B}$:

- $[\![acyclic(r)]\!](G)$ gdw. $[\![r]\!](G)$ ist azyklisch
- $[\![irreflexive(r)]\!](G)$ gdw. $[\![r]\!](G)$ ist irreflexiv
- $[\![empty(r)]\!](G)$ gdw. $[\![r]\!](G) = \emptyset$

Vollständiges CAT erlaubt noch (sehr) viele weitere Konstrukte, die wir hier auslassen.
Ein wichtiges Feature benötigen wir noch:
CAT kann weitere Basisrelationen zu einer Candidate Execution hinzu fügen!
Das wichtigste Beispiel ist die sogenannte Coherence.

Def: Sei $G=(E, po, rf)$ eine Candidate Execution. Eine Coherence $co \subseteq Write(E) \times Write(E)$ ist eine partielle Ordnung aller Writes, sodass

- Writes auf dieselbe Variable total durch co geordnet sind
- Writes auf verschiedene Variablen ungeordnet sind.

Intuitive Idee von co:

- Parallele Threads haben ~~zwar~~ keine Ordnung untereinander und schicken parallel Writes an den Speicher
- Der Speicher empfängt mehrere Writes auf dieselbe Variable, kann diese aber nicht gleichzeitig verarbeiten.
  ↳ Der Speicher erzeugt beim Verarbeiten selber eine Ordnung
- Diese Intuition ist im Fall von Multi-Speichern nicht ganz korrekt, aber selbst dort findet Coherence als Konzept Verwendung

In der Praxis nutzt jedes Speichermodell co, daher fügen wir sie direkt zu unserer Sprache als Basisrelation hinzu:

base ::= ... | co

Def (MM durch CAT):

Sei C eine CAT File. C definiert ein Speichermodell $MM_C : CExec \to \mathbb{B}$ mit

$$MM_C(G=(E,po,rf)) \Leftrightarrow \exists co: \forall a \in C: [\![a]\!](G'=(E,po,rf,co))$$

(∃co: Cherence; ∀a∈C: Axiom)

Bsp (SC):

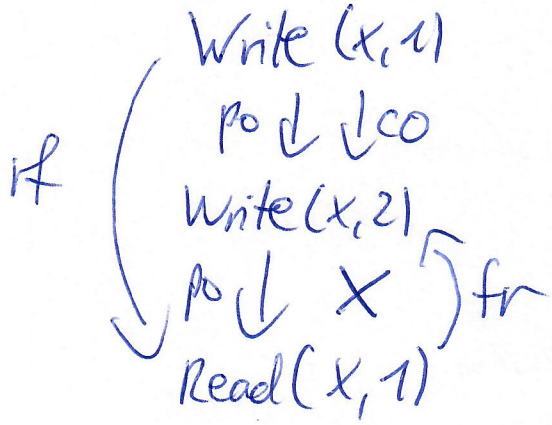
CAT:

```
let fr := rf^-1 ; co   // Konfliktrelation
let com := rf ∪ co ∪ fr
let ghb := po ∪ com    // global-happens-before
acyclic (ghb)
```

fr = from-read Relation (auch Konfliktrelation).

~~Diese drückt aus,~~

Diese wird benutzt um auszudrücken, dass ein Read keine veralteten Werte liest:



Bem: Wenn ghb acydic ist, dann kann ghb linearisiert, d.h. zu einer totalen Ordnung gemacht, werden.
Jede Linearisierung entspricht einem Interleaving der Events, welches alle die selbe Semantik aufweise

Nun versuchen wir TSO zu modellieren:

(Version 1):

```
let com-tso := rf ∪ co ∪ fr
let po-tso  := po \(W×R)   // Buffereffekt
let ghb := po_tso ∪ com-tso
acyclic (ghb)
```

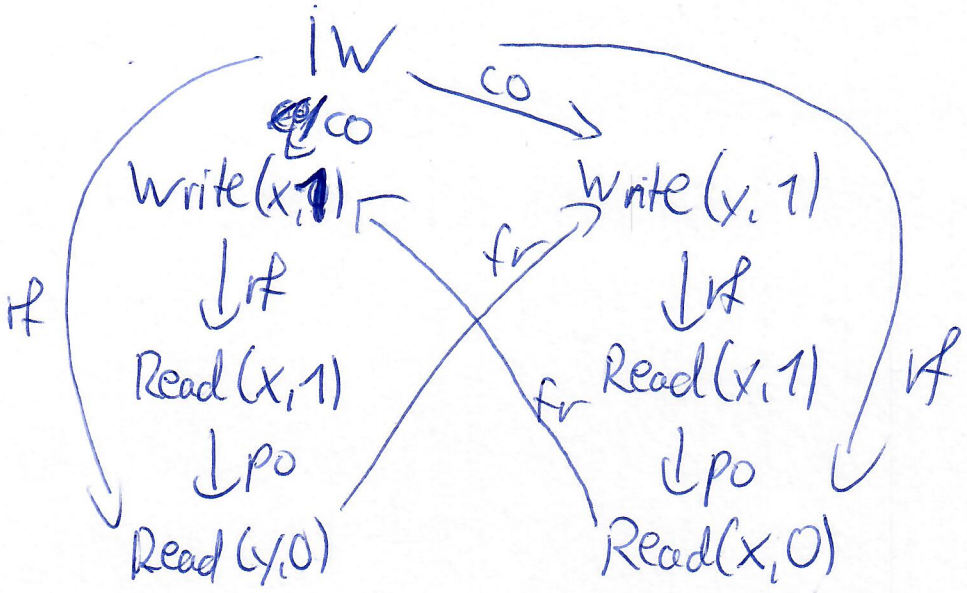
Buffereffekt:

(write) x := 5;
(Read) R := y;

⇝ Write(x,5) kann im Speicher ankommen, bevor Read(x,?) passiert.
Durch den Store-Buffer kann Write(x,5) auch erst nach dem Read im Speicher landen ▽

Problem: Reads erzeugen immer globale Ordnung, auch wenn diese early Reads sind (= aus dem Buffer lesen)

Bsp:



Dies sollte TSO erlauben !?

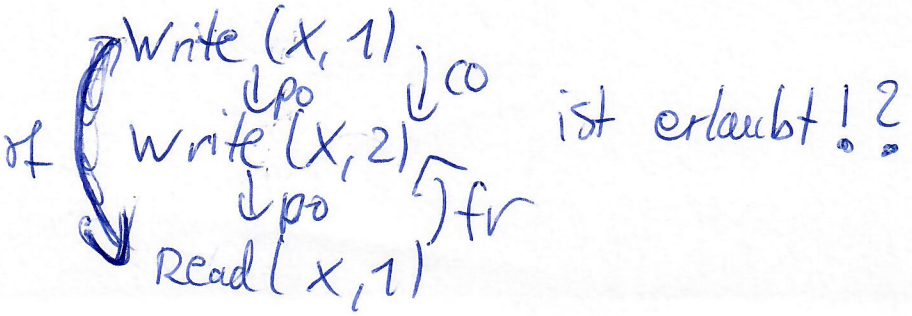
Version 2 :

```
let com-tso = (rfe ∩ ext) ∪ co ∪ fr
let po-tso = po \ WxR
let ghb    = po-tso ∪ com-tso
acyclic (ghb)
```

Nur noch Reads von anderen Threads erzeugen Ordnung!?

Problem:

Wir haben gar keine Restriktionen auf ~~Reads~~ ~~innerhalb~~ lokalen Reads. Dies erlaubt es aus der Zukunft/Vergangenheit zu lesen!



ist erlaubt!?

Version 3

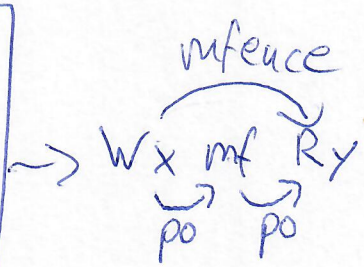
```
let com-tso := (rf ∩ ext) ∪ co ∪ fr
let po-tso := po \ (W × R)
let ghb = po-tso ∪ com-tso
acyclic (ghb)
acyclic( po ∪ (rf ∩ int)) // Keine Zukunftsreads
acyclic ( po ∪ (fr ∩ int)) // Keine Vergangenheitreads
```

Echte x86 Prozessoren haben einen "mfence" Befehl.
Dieser erzwingt das Leeren des Buffers.
Wir wollen dies auch modellieren können:

(1) Füge mfence-Befehl zur Syntax hinzu
(2) Erweitere SOS: mfence erzeugt neues Fence-Event
(3) Wir erweitern die Abstraktion G: Exec → CExec, sodass die Events E zusätzlich noch Fence-Events enthalten.
(4) Wir erweitern CAT um die Regel S ::= ... | F, mit der Semantik [[F]](G) = Fence(E).

TSO vollständig:

```
let com-tso = (rf ∩ ext) ∪ co ∪ fr
let mfence = (po ∩ (_ × F)); po
let po-tso = po \ (W × R) ∪ mfence
let ghb = po-tso ∪ com-tso
acyclic (ghb)
acyclic (po ∪ (rf ∩ int))
acylic (po ∪ (ff ∩ int))
```



Zum Abschluss: Auch Komplizierte Speichermodelle wie das C11 Speichermodell lassen sich in CAT ausdrücken:

- C11 hat verschiedene Access modes wie "SC, Release, Acquire, Relaxed und Non-Atomic"
- Um diese zu modellieren, erlaubt man "Tags" an Events.
- Ein Programm erzeugt nun getaggte Events
  Write/Read (x, val, tag)
- CAT wird erweitert, um auf getaggte Events zugreifen zu können

↳ In dieser Erweiterung lässt sich das C11 Speichermodell ausdrücken.