True Concurrency at Work: Relaxed Memory Models

Gérard Boudol
(joint work with Gustavo Petri)
INRIA Sophia Antipolis

Thiagu is 20

Thiagu is 20

Proof: born (to me) at the

Workshop on Computations of Distributed Systems, Königswinter, March 2-4, 1988.

Thiagu is 20

Proof: born (to me) at the

Workshop on Computations of Distributed Systems, Königswinter, March 2-4, 1988.

True vs “false” concurrency

Thiagu is 20

Proof: born (to me) at the

Workshop on Computations of Distributed Systems, Königswinter, March 2-4, 1988.

True vs "false" concurrency

A technique – equivalence by permutation of computations, [Berry & Lévy78] – to extract partial orders out of a standard operational semantics, i.e. transitions on configurations, with structure (syntax).

Plan of the talk:

- Motivation – shared variable concurrency
- Weak memory in hardware
- The problem
- Weak memory models
- Our approach: operational memory model
- Correctness (permutations of transitions)
- Perspectives and conclusion

Shared Variable Concurrency (1/3)

A renewal of interest in concurrent programming:

- new, naturally concurrent applications;
- multicore architectures.

Shared variable concurrency – aka multithreading – is difficult:

- state space explosion, non-determinism (testing, debugging?)
- data races, lack of atomicity... $\Rightarrow$ synchronization (mutual exclusion).

Shared Variable Concurrency (2/3)

Let $x{+}{+} = (x := !x + 1)$. Then, starting from $x = 0$ and $\mathsf{flag}_0 = tt = \mathsf{flag}_1$

$$\begin{array}{ll|ll} \mathsf{flag}_0 := \mathit{ff}\,; & & \mathsf{flag}_1 := \mathit{ff}\,; \\ \mathsf{if}\ !\mathsf{flag}_1\ \mathsf{then} & & \mathsf{if}\ !\mathsf{flag}_0\ \mathsf{then} \\ \quad x{+}{+} & & \quad x{+}{+}\,;\, x{+}{+} \end{array}$$

will return either $x = 1$ or $x = 2$.

Shared Variable Concurrency (3/3)

However, on most (multicore) machines

$$\begin{array}{lcl} \mathsf{flag}_0 := \mathit{ff}\,; & \| & \mathsf{flag}_1 := \mathit{ff}\,; \\ \mathsf{if}\ !\mathsf{flag}_1\ \mathsf{then} & & \mathsf{if}\ !\mathsf{flag}_0\ \mathsf{then} \\ \quad x++ & & \quad x++\,;x++ \end{array}$$

may return $x = \mathbf{3}$ – an unwanted outcome.

Shared Variable Concurrency (3/3)

However, on most (multicore) machines

$$\begin{array}{lcl} \mathsf{flag}_0 := \mathit{ff}\,; & \parallel & \mathsf{flag}_1 := \mathit{ff}\,; \\ \mathsf{if}\ !\mathsf{flag}_1\ \mathsf{then} & & \mathsf{if}\ !\mathsf{flag}_0\ \mathsf{then} \\ \quad x{+}{+} & & \quad x{+}{+}\,;\,x{+}{+} \end{array}$$

may return $x = \mathbf{3}$ – an unwanted outcome.

Why? A write to the memory is issued, and the computation continues without waiting for it to be performed – think of

$$\begin{array}{ccccccc} \mathsf{flag}_0 := \mathit{ff} & \parallel & \mathsf{if}\ !\mathsf{flag}_1\ \mathsf{then} & \parallel & \mathsf{flag}_1 := \mathit{ff} & \parallel & \mathsf{if}\ !\mathsf{flag}_0\ \mathsf{then} \\ & & x{+}{+} & & & & x{+}{+}\,;\,x{+}{+} \end{array}$$

Weak Memory in Hardware (1/2)

Optimization for sequential code: writes to the memory run in parallel with the code.

➥ reordering of instructions:

$$x := 1\,;\, r := !y$$

may appear to be executed as

$$r := !y\,;\, x := 1$$

But program order on read/writes on a given memory location must be preserved:

$$x := 1\,;\, x := 2\,;\, r := !x$$

must return $r = 2$.

➥ write buffers (and caches). Also: delayed reads.

WEAK MEMORY in HARDWARE (2/2)

Synchronization:

- test-and-set, compare-and-swap...
- memory barriers, fence...

↪ locks ℓ to ensure mutual exclusion (**with** ℓ **do** $\cdots$).

For instance

$$(\textbf{with}\ \ell\ \textbf{do}\ x++) \parallel (\textbf{with}\ \ell\ \textbf{do}\ x++)$$

returns $x = 2$, whereas $x++ \parallel x++$ may return $x = 1$:

$$\begin{aligned} x++ \parallel x++ &\rightarrow x := 0+1 \parallel x := !x+1 \\ &\xrightarrow{*} x := 1 \parallel x := !x+1 \\ &\xrightarrow{*} x := 1 \parallel x := 1 \end{aligned}$$

a data race.

An Alternative

Multithreading does not work on optimized, multicore architectures.

- ➥ give up optimizations – should hardware factories build (and sell) non-optimized machines?
- ➥ program in a disciplined way: write only (concurrent) programs that work!

Using synchronization/memory barriers.

A Challenge

How to be sure that any "well-synchronized" program works on a relaxed architecture?

- formalize the "memory model,"
- show that it implements the usual, interleaving semantics for well-synchronized (= Data Race Free) programs, that is

prove the **"DRF guarantee."**

Weak Memory Models (1/4)

Intended to formalize the semantics supported by optimized hardware (and, sometimes, compilers).

- specify which value a read instruction may (or not) get during execution;
- delimit which optimizations (reorderings) are allowed;
- should support the DRF guarantee.

Weak Memory Models (2/4)

Intel 64 (2007)

Stores are not reordered with older loads:

Processor 0	Processor 1
`mov r1, [_ x] // M1` `mov [_ y], 1 // M2`	`mov r2, [_ y] // M3` `mov [_ x], 1 // M4`
Initially `x == y == 0` `r1 == 1` and `r2 == 1` is not allowed	

The JMM (Java Memory Model):

Initially `x == y == 0`

Thread 1	Thread 2
`r1 = x`	`r2 = y`
`y = 1`	`x = 1`

Allowed: `r1 == 1 == r2`

Weak Memory Models (3/4)

JAVA Memory Model [Sevcik & Aspinall 2008]:

- action $a = (t, k, u)$ where t = thread identifier, k = kind (read, write,...), u = unique identifier.
- execution $E = (P, \leq_{po}, \leq_{so}, V, W)$ where
  - P = program = set of traces (sequences of actions);
  - $\leq_{po}$ program order on actions;
  - $\leq_{so}$ synchronization order on actions;
  - V the visibility mapping, from read actions to write actions;
  - W written value assigned to each (write) action.
- happens-before order $\leq_{hb} = (\leq_{sw} \cup \leq_{po})^+$ where $\leq_{sw}$ is...

Weak Memory Models (4/4)

An execution $E = (P, \leq_{po}, \leq_{so}, V, W)$ is well-formed iff

1. ...

$\vdots$

4. $\leq_{so}$ is consistent with $\leq_{po}$, that is ...;

$\vdots$

9. $\leq_{hb}$ is consistent with V, that is ...;
10. ...

A well-formed execution is legal iff ... (7 axioms).

➥ more than 3 (LNCS) pages in [Sevcik & Aspinall 2008] to define the "allowable behaviour" of a program.

Problem

How to prove the DRF guarantee for such a memory model?

- how to relate the usual, interleaving semantics of a program (= expression written in some programming language) with such a memory model?
- how to define the semantics of a program determined by the memory model?

Looks like a "true concurrency problem," but...

- a "truly concurrent" version of the interleaving semantics would leave the input/output behaviour invariant, thus cannot account for the "weak semantics" (unexpected outcomes).

➥ A new approach to formalizing relaxed memory models.

Our Approach

Considering a memory model, to establish the DRF guarantee for a given language, with threads and locks:

- define the "reference" interleaving semantics (as usual),
- define the memory model as part of a weak operational semantics;
- prove that the weak semantics implements the interleaving semantics for DRF programs.

We use a bisimulation method, and we need to establish that DRF programs are well-synchronized.

- proved using "true-concurrency" techniques (Berry & Lévy's permutation equivalence).

Language

An ML-like language (no typing), with threads and locks:

$$
\begin{array}{llll}
e & ::= & v \mid (e_0 e_1) & \textit{expressions} \\
 & \mid & (\text{ref } e) \mid (!\, e) \mid (e_0 := e_1) & \\
 & \mid & (\text{thread } e) \mid (\text{with } \ell \text{ do } e) & \\
v & ::= & x \mid \lambda x e \mid () & \textit{values} \\
\mathbf{E} & ::= & [] \mid (\mathbf{E}\, e) \mid (v\, \mathbf{E}) & \textit{evaluation contexts} \\
 & \mid & (\text{ref } \mathbf{E}) \mid (!\, \mathbf{E}) \mid (\mathbf{E} := e) \mid (v := \mathbf{E}) & \\
 & \mid & (\text{holding } \ell \text{ do } \mathbf{E}) &
\end{array}
$$

equipped with an interleaving semantics.

Sequential composition:

$$e_0 \,;\, e_1 =_{\text{def}} (\text{let } x = e_0 \text{ in } e_1) = (\lambda x e_1 e_0)$$

(x not free in e_1).

Write Buffers

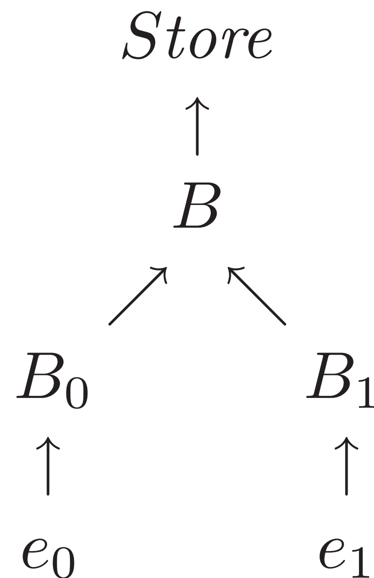
Write buffer: a queue of values (FIFO) per pointer.

$$B = \{p_1 \mapsto v_1^1 \cdots v_{n_1}^1 \ , \ \ldots \ , \ p_k \mapsto v_1^k \cdots v_{n_k}^k\}$$

Syntax for thread systems:

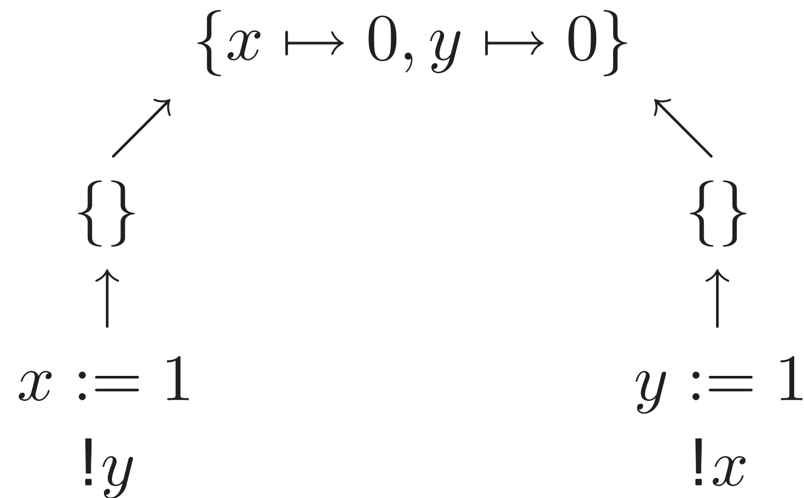
$$\Theta ::= e \ \mid \ \langle B \rangle \Theta \ \mid \ (\Theta \parallel \Theta)$$

depicted as, e.g.



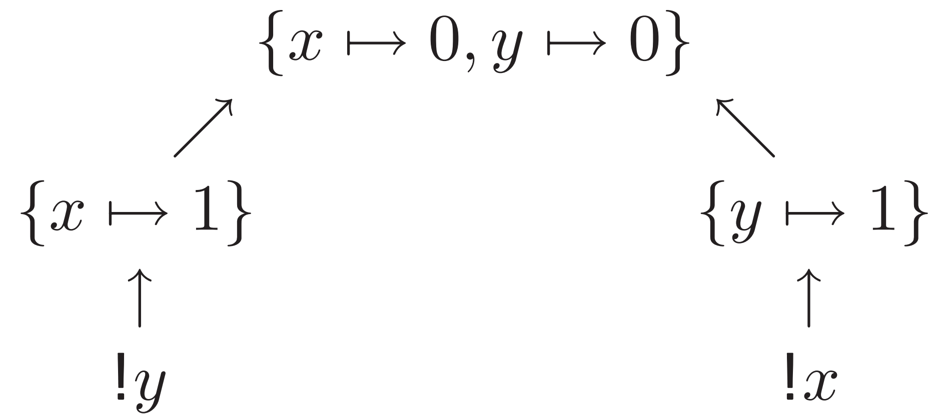
WEAK SEMANTICS – ROUGHLY (1/3)

$$x := 1\,;\,!y \;\parallel\; y := 1\,;\,!x$$



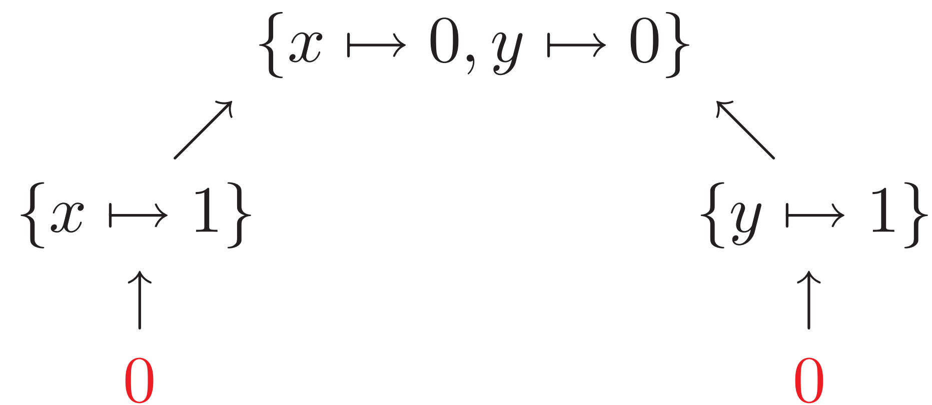
Weak Semantics – roughly (2/3)

$$x := 1\ ;\ !y \parallel y := 1\ ;\ !x$$



Weak Semantics – roughly (3/3)

$$x := 1\,;\,!y \;\|\; y := 1\,;\,!x$$



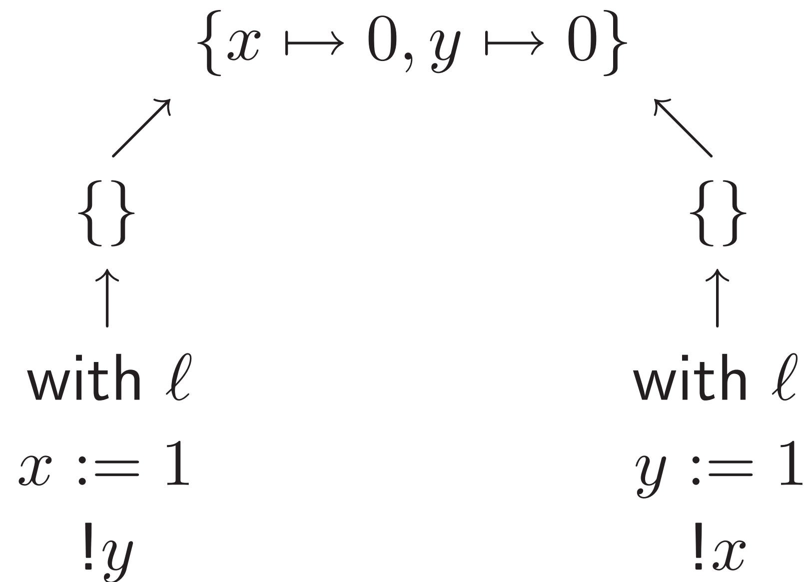
Weak Semantics – formally

Main rules: read, write and unlock + propagation of writes.

$$
\begin{array}{rcll}
(S, L, \boldsymbol{\Theta}[\mathbf{E}[(!\,p)]]) & \to & (S, L, \boldsymbol{\Theta}[\mathbf{E}[v]]) & (S, \boldsymbol{\Theta})(p) = v \\
(S, L, \boldsymbol{\Theta}[\mathbf{E}[(p := v)]]) & \to & (S, L, \boldsymbol{\Theta}[\langle\{p \mapsto v\}\rangle \mathbf{E}[()]]) & \\
(S, L, \boldsymbol{\Theta}[\mathbf{E}[(\mathsf{holding}\ \ell\ \mathsf{do}\ v)]]) & \to & (S, L - \{\ell\}, \boldsymbol{\Theta}[\mathbf{E}[v]]) & \boldsymbol{\Theta}\dagger \\
(S, L, \langle B \rangle \Theta) & \to & (S[p := v], L, \langle B \uparrow p \rangle \Theta) & \\
 & & B(p) = v \cdot s & \\
(S, L, \boldsymbol{\Theta}[\langle B_0 \rangle \langle B_1 \rangle \Theta]) & \to & (S, L, \boldsymbol{\Theta}[\langle B_0[p \leftarrow v] \rangle \langle B_1 \uparrow p \rangle \Theta]) & \\
 & & B_1(p) = v \cdot s & \\
(S, L, \boldsymbol{\Theta}[(\langle B \rangle \Theta \parallel \Theta')]) & \to & (S, L, \boldsymbol{\Theta}[\langle\{p \mapsto v\}\rangle (\langle B \uparrow p \rangle \Theta \parallel \Theta')]) & \\
 & & B(p) = v \cdot s & \\
(S, L, \boldsymbol{\Theta}[(\Theta \parallel \langle B \rangle \Theta')]) & \to & (S, L, \boldsymbol{\Theta}[\langle\{p \mapsto v\}\rangle (\Theta \parallel \langle B \uparrow p \rangle \Theta')]) & \\
 & & B(p) = v \cdot s & \\
(S, L, \boldsymbol{\Theta}[\langle B \rangle \Theta]) & \to & (S, L, \boldsymbol{\Theta}[\Theta]) & \mathsf{W}(B) = \emptyset
\end{array}
$$

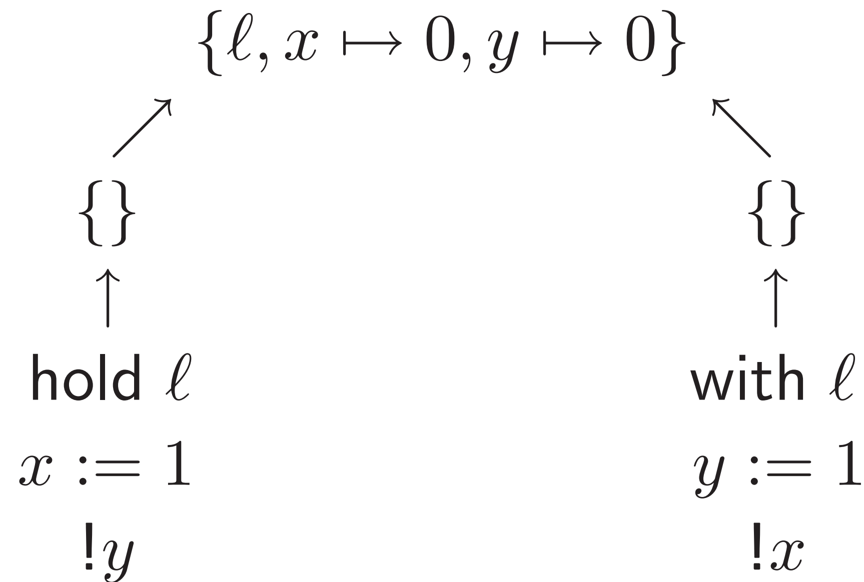
Weak Semantics: Example (1/5)

$$(\text{with } \ell \text{ do } x := 1)\ ;\ !y \ \parallel\ (\text{with } \ell \text{ do } y := 1)\ ;\ !x$$



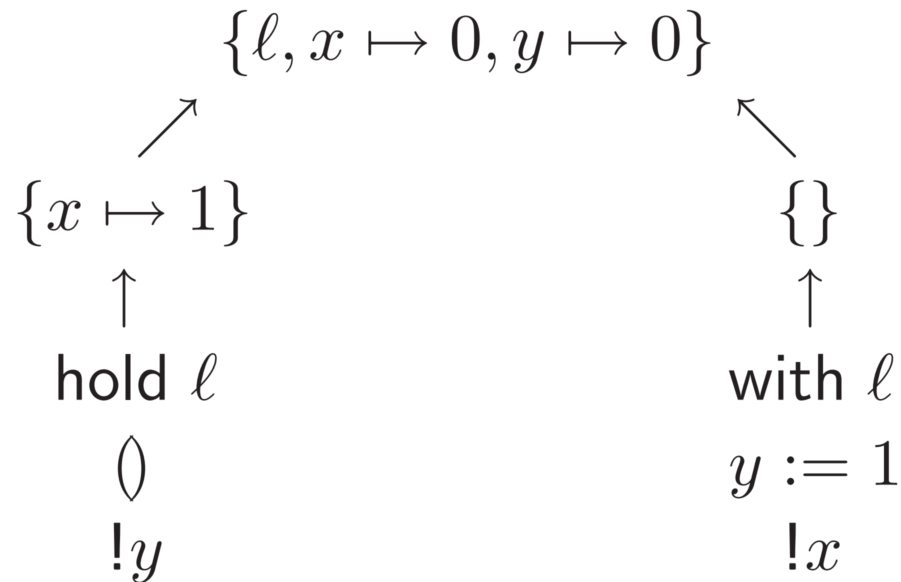
Weak Semantics: Example (2/5)

(with ℓ do $x := 1$) ; !y $\|$ (with ℓ do $y := 1$) ; !x



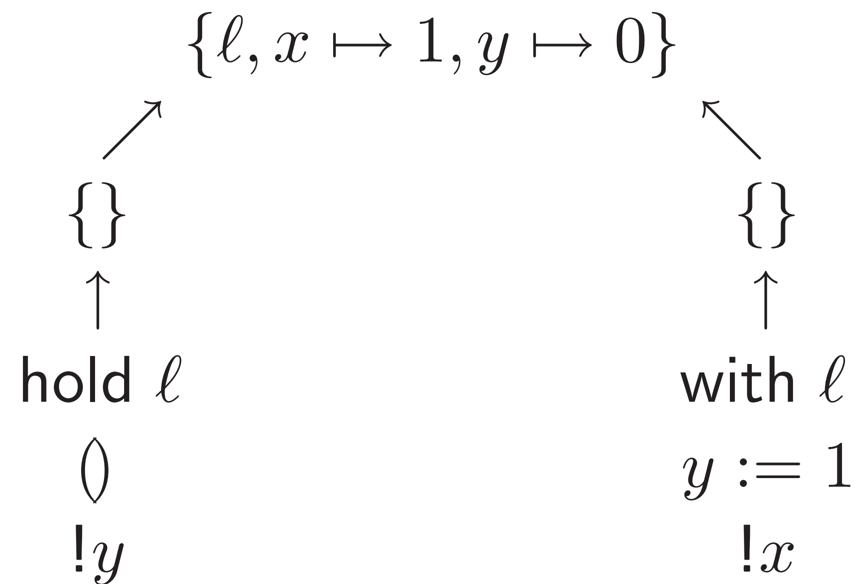
Weak Semantics: Example (3/5)

$$(\text{with } \ell \text{ do } x := 1) \; ; \; !y \;\; \| \;\; (\text{with } \ell \text{ do } y := 1) \; ; \; !x$$



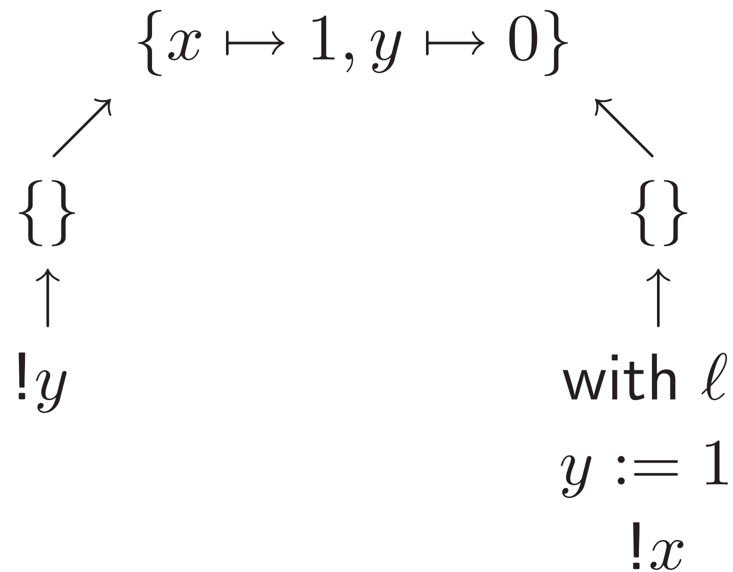
Weak Semantics: Example (4/5)

$$(\text{with } \ell \text{ do } x := 1) \,;\, !y \;\|\; (\text{with } \ell \text{ do } y := 1) \,;\, !x$$



Weak Semantics: Example (5/5)

$(\text{with } \ell \text{ do } x := 1) \; ; \; !y \;\; \| \;\; (\text{with } \ell \text{ do } y := 1) \; ; \; !x$



Correctness

- standard, or strong configuration: $C = (S, L, \Theta)$ where Θ does not contain any write buffer – otherwise: weak configuration.
- DRF configuration: cannot reach (in the interleaving semantics) a configuration C where there are conflicting, concurrent accesses to the same pointer, i.e. $C = (S, L, \Theta)$ where

$$\Theta = \cdots \parallel \mathbf{E}[e] \parallel \cdots \parallel \mathbf{E}[e'] \parallel \cdots$$

where e and e' are reads $(!p)$ or writes $(p := v)$ to the same pointer, one of which is a write.

Theorem. The strong configurations reachable from a (strong) DRF configuration C in the weak semantics coincide with the configurations reachable from C in the interleaving semantics.

Some Notations

- transitions in the weak semantics: $C \rightarrow C'$.
- propagation of buffered writes: $C \rightarrow_b C'$.
- transitions in the interleaving semantics (C strong): $C \rightarrow_s C'$.

BISIMULATION (1/2)

Lemma. For any configuration C there exists a strong configuration C' such that $C \xrightarrow{*}_b C'$. Notation: $C \Downarrow C'$.
Proof: propagation of buffered writes terminates.

C a strong configuration, $\mathcal{R}(C)$ given by

$$C_0 \, \mathcal{R}(C) \, C_1 \quad \Leftrightarrow_{\text{def}} \quad C \xrightarrow{*} C_0 \Downarrow C_1$$

The weak semantics simulates the interleaving semantics:

$$\begin{array}{ccccc} C & \xrightarrow{*} & C_0 & \rightarrow & C_0' \\ & & \Downarrow & & \Downarrow \\ & & C_1 & \rightarrow_s & C_1' \end{array}$$

Coherence

- coherent configuration: no concurrent buffered writes on the same pointer.
- fully coherent configuration: in addition, no buffered write concurrent with a read on the same pointer.

Fully coherent $\approx$ without "weak data race."

Lemma. Buffered writes propagation (i.e. $\rightarrow_b$) preserves full coherence, and, for coherent configurations, is locally confluent.

Corollary. (Confluence) C coherent:

$$C \Downarrow C_0 \;\&\; C \Downarrow C_1 \;\Rightarrow\; C_0 = C_1$$

Bisimulation (2/2)

Claim. C DRF & $C \xrightarrow{*} C'$ $\Rightarrow$ C' fully coherent.

C DRF: the weak semantics does not deviate from the interleaving one.

$$\begin{array}{ccccc} C & \xrightarrow{*} & C_0 & \longrightarrow & C'_0 \\ & & \Downarrow & & \\ & & C_1 & & \end{array} \quad \Rightarrow \quad \begin{array}{c} C'_0 \\ \Downarrow \\ C_1 \end{array} \quad \text{or} \quad \begin{array}{ccc} & & C'_0 \\ & & \Downarrow \\ C_1 & \longrightarrow_s & C'_1 \end{array}$$

➥ **correctness.**

The Claim – roughly

- a strong configuration C is well-synchronized if in any strong computation of C, two conflicting, concurrent actions are separated by an unlock in the same thread as the one of the first action.

Proposition. If C is DRF then C is well-synchronized.

Lemma. C strong. If $C \xrightarrow{*} C'$, where C' contains a write buffered for pointer p, then in the reduction there is a write $(p := v)$ on the same thread to the store, which is not followed by any unlock (unless concurrent with the write).

Corollary. C DRF & $C \xrightarrow{*} C' \Rightarrow C'$ fully coherent.

DRF $\Rightarrow$ Well-Synchronized (1/2)

(now we consider only the interleaving semantics, and strong configurations.)

C DRF:

$$C \xrightarrow{*} \xrightarrow[t]{a} \underbrace{\cdots}_{n} \xrightarrow[t']{b} \cdots$$

where a and b are conflicting ($a \# b$), concurrent ($t \neq t'$) actions, then, for $n = 0$:

- a and b cannot be accesses to the same pointer, one of them a write (would contradict DRF);
- if a and b are "acquire" or "release" of the same lock, a must be a release (= unlock).

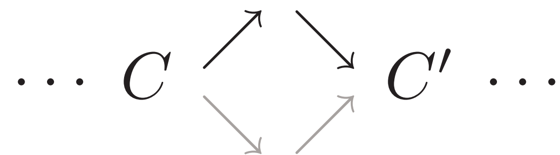
For $n > 0$? By cases on the step following a, possibly commuting the two – permutation of transitions.

PERMUTATION of TRANSITIONS

In the interleaving semantics, concurrent, non-conflicting steps can be permuted:

$$C \xrightarrow[t]{a} \xrightarrow[t']{b} C' \;\&\; \neg(a \# b) \;\&\; t \neq t' \;\Rightarrow\; C \xrightarrow[t']{b} \xrightarrow[t]{a} C'$$

➥ generates an equivalence on computations



Event = occurrence of an action in such a computation.

➥ ordering of events: e "happens before" e' in a computation iff e precedes e' in any permutation of the computation.

DRF $\Rightarrow$ Well-Synchronized (2/2)

Crucial Lemma. For any computation of a DRF program, if an event e performed by thread t happens before e', performed by thread t', in the computation, while being concurrent with e' $(t \neq t')$, then t performs an unlock operation that happens before e' in the same computation.

Proof: transposition of computing steps between e and e' – details in the paper.

Work in Progress

- specialize the model, to make it closer to (less relaxed) specific architectures: Itanium, ARM, SPARC's TSO and PSO.
- extend it to capture more relaxed architectures, with delayed reads: SPARC's RMO, Power PC, Intel 64.
- make a similar study for speculative computation, where evaluation order is completely relaxed:
  - guessing (instead of reading) values from the memory, and
  - computing in advance, e.g. in the branches of a conditional construct and the continuation of a sequential composition.

Conclusion

- True concurrency techniques are appropriate to study the semantics of concurrent programs running on truly concurrent machines.
- Weak operational models may be useful for reasoning about programs.
- **Big challenge:** back-end compiler's optimizations.
- Is (truly) shared memory a good idea?

Happy Birthday, Thiagu!