A Denotational Semantics for Weak Memory Concurrency

Stephen Brookes

Carnegie Mellon University
Department of Computer Science

Midlands Graduate School
April 2016

Work supported by NSF

Motivation

- Concurrent programs are everywhere.
  Relied on for efficiency and correctness:
  - databases, phones, banks, industry, autonomous vehicles

- Ensuring correctness of concurrent programs is hard:
  - "The major problem facing software developers. . . "

  Behavior depends on whether threads cooperate or interfere.

- We need formal methods and automated tools.
  And we would benefit from *compositional reasoning*:
  - Exploit modularity, focus on local analysis, reduce complexity

- Big gap between theory and practice.
  - Modern hardware is incompatible with traditional semantics.
  - Current tools lack generality and scalability.

Foundational research is essential.

Themes

Shared-memory parallel programs

- concurrent threads or processes
- reading and writing to shared state
- using locks to avoid data races

Denotational semantics

- choose an appropriate semantic domain
  - *abstract, but computationally accurate*
  - *tailored to program behavior*
- syntax-directed (compositional) semantic clauses
  - *structural induction*
  - *recursion = fixed-point*

Past, present, future

- status quo, limitations and defects
  - *historical context*
- new ideas and directions
  - *further research*

Compositionality Principle

from Wikipedia

> ***... the meaning of a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them.***

- Also called Frege's principle, because G. Frege (1848-1925) is widely credited with the first modern formulation.
- However, the idea appears among early Indian philosophers and in Plato's work.
- Moreover the principle was never explicitly stated by Frege, and was assumed by Boole decades earlier.

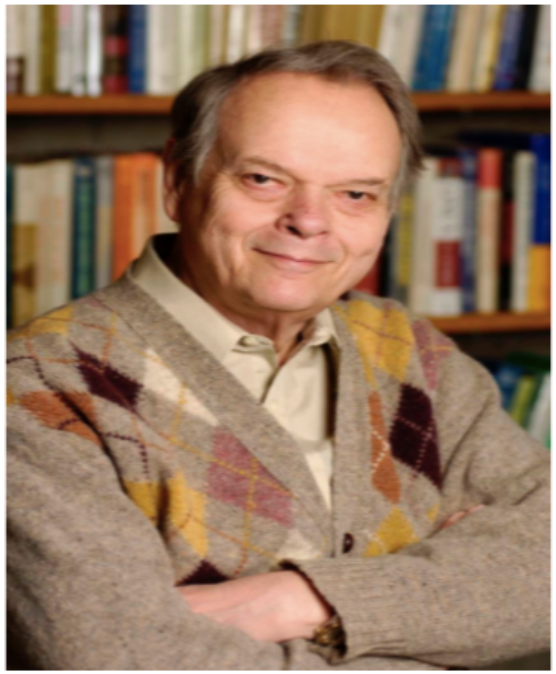
  Boole, G. (1854). *An investigation of the laws of thought*

Serves as a methodological principle to guide the development of theories of syntax and semantics.

The characteristic principle of *denotational semantics* and of *structural operational semantics*.

The Art of Denotational Semantics

- The right foundations (a good choice of denotation) and the right definitions (encapsulation of key concepts) should lead naturally to the right theorems.
- The right development (a good choice of notation) should help, not hinder, program design and analysis.
- Not as easy as it sounds, especially for concurrent programs. . .



John Reynolds (1935-2013)

Prologue

"Of the many forms of false culture, a premature converse with abstractions is perhaps the most likely to prove fatal . . ." *George Boole (1859)*

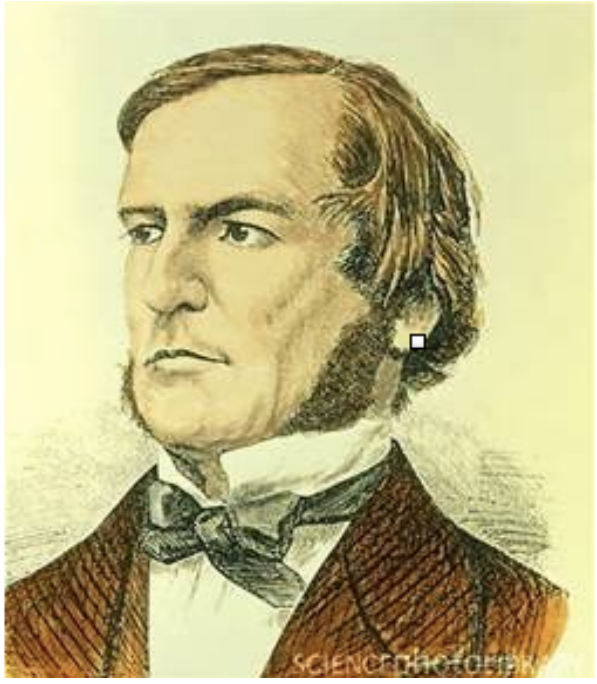
AN INVESTIGATION

OF

THE LAWS OF THOUGHT,

ON WHICH ARE FOUNDED

THE MATHEMATICAL THEORIES OF LOGIC AND PROBABILITIES.



Shared-memory programs

- ***Threads*** or processes, reading and writing to shared state
  - threads may be sequential (simple case)
  - threads may *fork* and *join* (nested parallelism)
  - may have private local variables

- ***Race condition***, causing unpredictable behavior,
  when one thread writes to a variable being used by another

$$x{:=}x+1 \parallel y{:=}x+x$$

- ***Synchronization primitives*** such as

  locks, conditional critical regions, compare-and-swap, ...

  can be used to ensure mutually exclusive access

$$(\textbf{lock}\ r;\ x{:=}x+1;\ \textbf{unlock}\ r) \| (\textbf{lock}\ r;\ y{:=}x+x;\ \textbf{unlock}\ r)$$

Denotational Semantics

Shared-memory programs

Historically, concurrency is viewed as ***difficult to deal with***

- early approaches limited to "simple" programs (no heap)
- issues such as *fairness* and *unbounded non-determinism*

Need a suitably abstract semantic domain

- not tied to specific hardware
- based on machine-independent view of "state"
- abstracting away from thread id's and schedulers
- yet concrete enough to yield *accurate* information

- Want semantics to support *compositional reasoning* about *program properties*

  in any reasonable implementation...

  (while avoiding implementation details)

Program Properties

- **Partial correctness** $\{p\}c\{q\}$
  Every terminating execution of c from a state satisfying p ends in a state satisfying q.
- **Total correctness**
  Every execution of c from a state satisfying p terminates, and ends in a state satisfying q.
- **Safety**
  Something bad never happens,
  e.g. "In every execution of c, the value of x never decreases."
- **Liveness**
  Something good eventually happens,
  e.g. "In every execution of c, x is eventually set to 1."

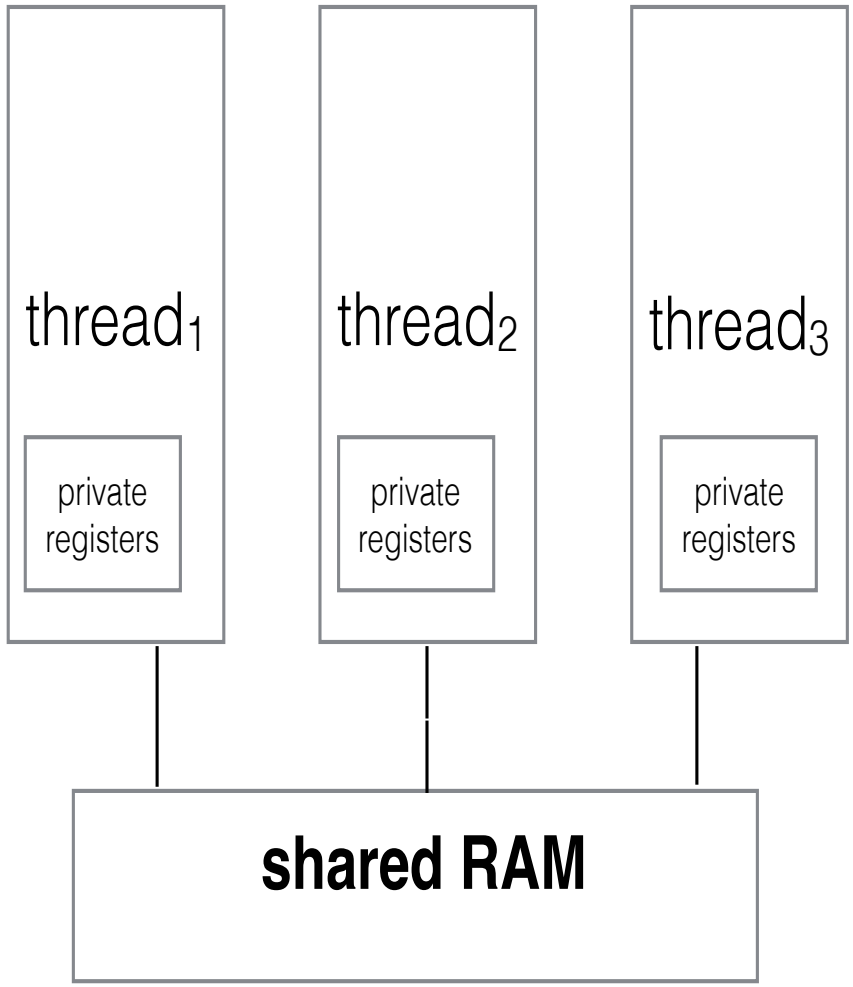
Sequential Consistency

Traditional semantics for shared-memory programs assume ***sequential consistency*** (SC)

> *"The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each processor appear in the order specified by its program."*
>
> *Leslie Lamport (1979)*

- Instructions are executed in program order.
- Each write operation becomes instantly visible to all threads.
- As if running on a uniprocessor...

Abstract View



SC Semantics

- Assuming SC leads naturally to models based on ***global states***, ***traces*** and ***interleaving***.

- Can give a ***denotational*** semantics, in which:
  - programs denote sets of ***traces***
  - traces are finite or infinite sequences of ***actions***, allowing for "environment" interaction
  - parallel composition is ***fair interleaving*** *Park 1979*

$$\mathcal{T}(c_1 \| c_2) = \bigcup \{ \alpha_1 \| \alpha_2 \mid \alpha_1 \in \mathcal{T}(c_1) \ \& \ \alpha_2 \in \mathcal{T}(c_2) \}$$

- Can also give an ***operational*** semantics, in which:
  - states are global, steps are atomic actions

$$\frac{\langle c_1, \sigma \rangle \to \langle c_1', \sigma' \rangle}{\langle c_1 \| c_2, \sigma \rangle \to \langle c_1' \| c_2, \sigma' \rangle} \qquad \frac{\langle c_2, \sigma \rangle \to \langle c_2', \sigma' \rangle}{\langle c_1 \| c_2, \sigma \rangle \to \langle c_1 \| c_2', \sigma' \rangle}$$

Compositionality

- Program properties such as

  partial and total correctness, safety and liveness

  are based on (interference-free) execution, e.g.

$$\{p\}c\{q\} : \ \forall \alpha \in \mathcal{T}(c).\ \text{if } \sigma \models p \ \&\ \sigma \stackrel{\alpha}{\Longrightarrow} \sigma' \text{ then } \sigma' \models q$$

- Fairness crucial for liveness; infinite traces for safety, liveness.
- These properties involve *sequentially executable* traces.
- To determine the (sequential) traces of $c_1 \| c_2$,
  need to include non-sequential traces of c_1 and c_2.

  The sequential traces of $c_1 \| c_2$ are not always obtained
  by interleaving sequential traces of c_1 and c_2.
- So $\mathcal{T}(c)$ includes non-sequential traces and we can define

$$\mathcal{T}(c_1 \| c_2) = \bigcup \{\alpha_1 \| \alpha_2 \mid \alpha_1 \in \mathcal{T}(c_1) \ \&\ \alpha_2 \in \mathcal{T}(c_2)\}$$

To support compositional reasoning,

we must allow for interaction with environment...

Advantages

- A simple action trace semantics supports compositional reasoning about simple shared-memory programs

$$\frac{\{p_1\}c_1\{q_1\} \quad \{p_2\}c_2\{q_2\}}{\{p_1 \wedge p_2\}c_1 \| c_2\{q_1 \wedge q_2\}}$$

Owicki, Gries 1976

- An action trace semantics incorporating heap, race detection and resource-awareness serves as foundation for ***Concurrent Separation Logic***

$$\frac{\{p_1\}c_1\{q_1\} \quad \{p_2\}c_2\{q_2\}}{\{p_1 \star p_2\}c_1 \| c_2\{q_1 \star q_2\}}$$

O'Hearn, Brookes 2007

- $\{p\}c\{q\}$ interpreted as "in all (suitable) environments. . . "
  - *rely/guarantee* trade-off between process and environment
  - *local reasoning*, *separability* and *ownership transfer*
- **Provability implies** *no races*.

Historical Snapshots

Trace-based denotational semantics have been widely used, for *shared memory* and for *channel-based communication*:

- Park: steps (σ, σ') as atomic assignments [1979]
  - fixed-point characterization of *fairmerge*
- Hoare: steps $h?v$, $h!v$ as communication events [1983]
  - led to *failures/divergences*, FDR model checker for CSP
- B: steps (σ, σ') as finite sequences of actions [1993]
  - *transition traces* with *stuttering* and *mumbling*
  - simpler characterization of *fairmerge*
  - *fully abstract* w.r.t observing histories
- B: fair communicating processes [2002]
- B: steps as store and heap operations, race detection [2007]
  - soundness of Owicki-Gries
  - soundness of Concurrent Separation Logic, permissions, ...

Limitations and Defects

- **Traces are simple**

  But it was surprisingly difficult to develop a tractable account of fairness!

- **Traces are *too* simple**

  Premature converse with an abstraction?

- **SC is "false concurrency"**

  parallel $\neq$ non-deterministic interleaving

- **SC is impractical**

  "...achieving sequential consistency may not be worth the price of slowing down the processors."

  Leslie Lamport (1979)

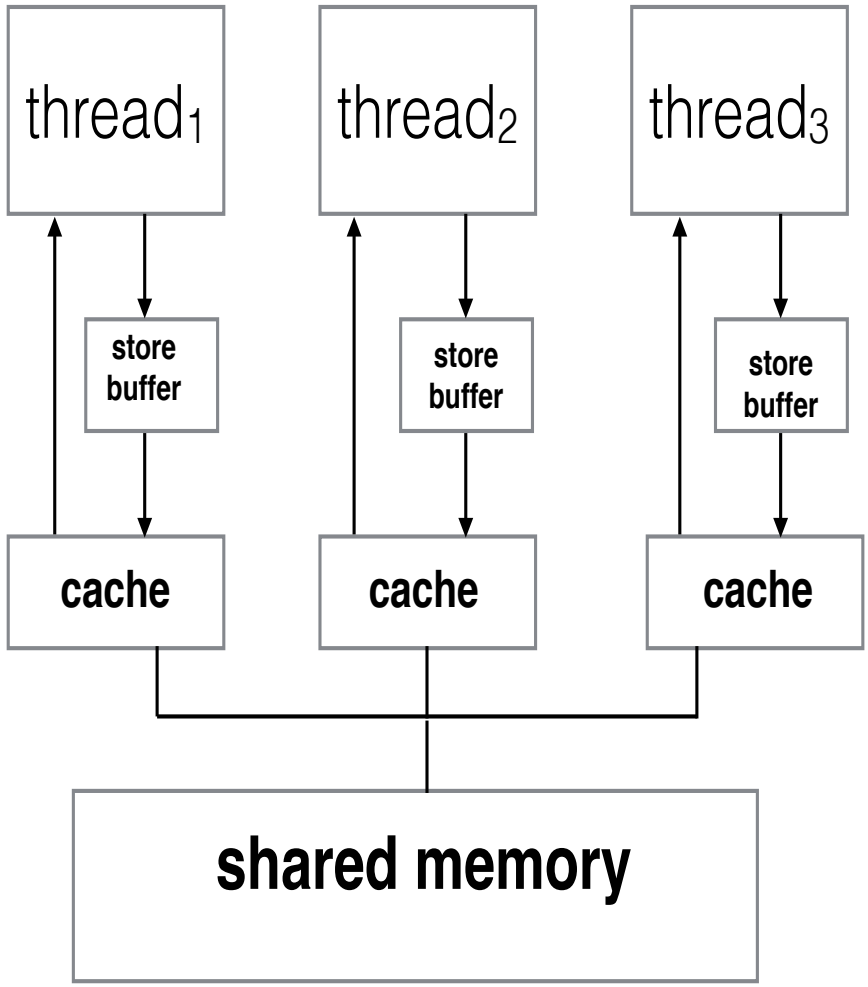
- ***Trace semantics* is only appropriate for SC**

Reality

- Modern multi-processors provide memory models with ***weaker guarantees*** and ***faster results***
  - reads may see stale values, because of buffering or caches
  - memory operations may get re-ordered, for optimization
  - there may be no persistent "global" state
  - results may be inconsistent with SC
- All processors are not equal
  - ARM, x86, Power, Sparc, ...

  and they offer a range of memory models (stronger to weaker)
  - SC, Total Store Order (TSO), release/acquire (C11),
- Mostly informal, unclear specifications.
- **Is your PC *really* SC?**

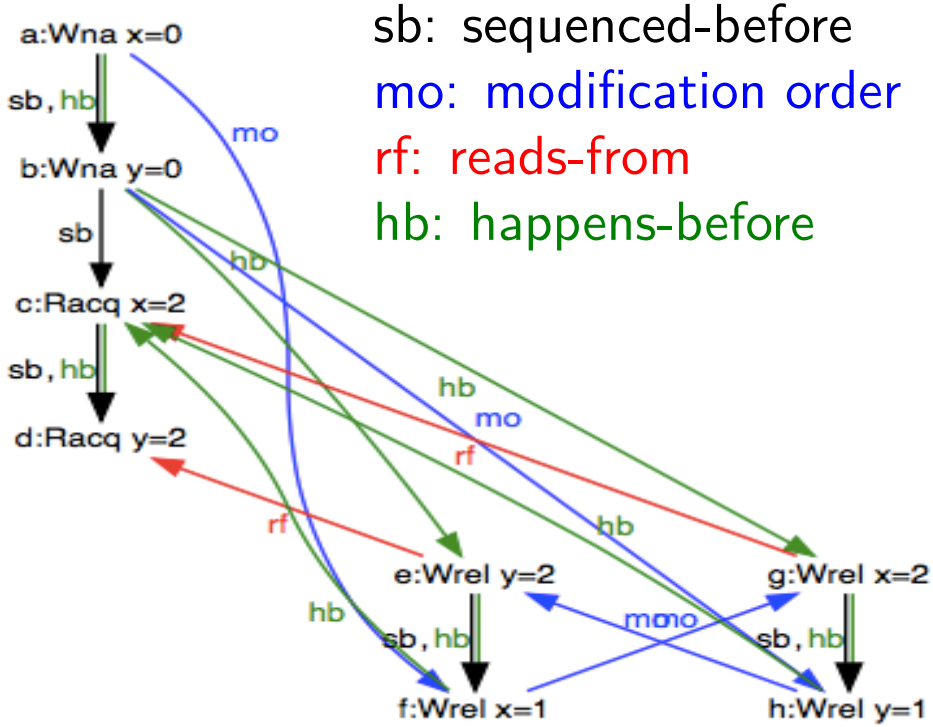
Architecture



Status Quo

Foundational work on C11 uses ***operational semantics***, ***execution graphs***, and ***weak memory axioms***

Alglave, Batty, Sewell, et al



$$(\text{CohRR}) : \neg\exists a, b.\ \mathrm{hb}(a, b) \ \wedge\ \mathrm{mo}(\mathrm{rf}(b), \mathrm{rf}(a))$$

Execution graphs

- An ***execution graph*** has ***nodes*** labelled with memory operations, and four kinds of relational ***edges***

$$(sb, mo, rf, hb)$$

that satisfy ***weak memory axioms***

sb is the "program order"
mo is a "modification order"
for each variable *i* a linear order on the writes to *i*
rf maps each *read* to the *write* it "reads from"
hb is the "happens before" relation

- The axioms impose *impossibility* constraints
  e.g. $(mo \cup rf^{-1} \cup sb)^+$ must have no cycles.
- An ***execution*** (σ, σ') is an execution graph whose initial reads are consistent with σ, and *mo*-final writes determine σ'.

Program Analysis

- To determine which weak memory executions are possible for a given program:

  Generate the possible execution graphs...
  ...and find edge relations constrained to obey the axioms.
  Extract the initial-final state pairs (σ, σ').

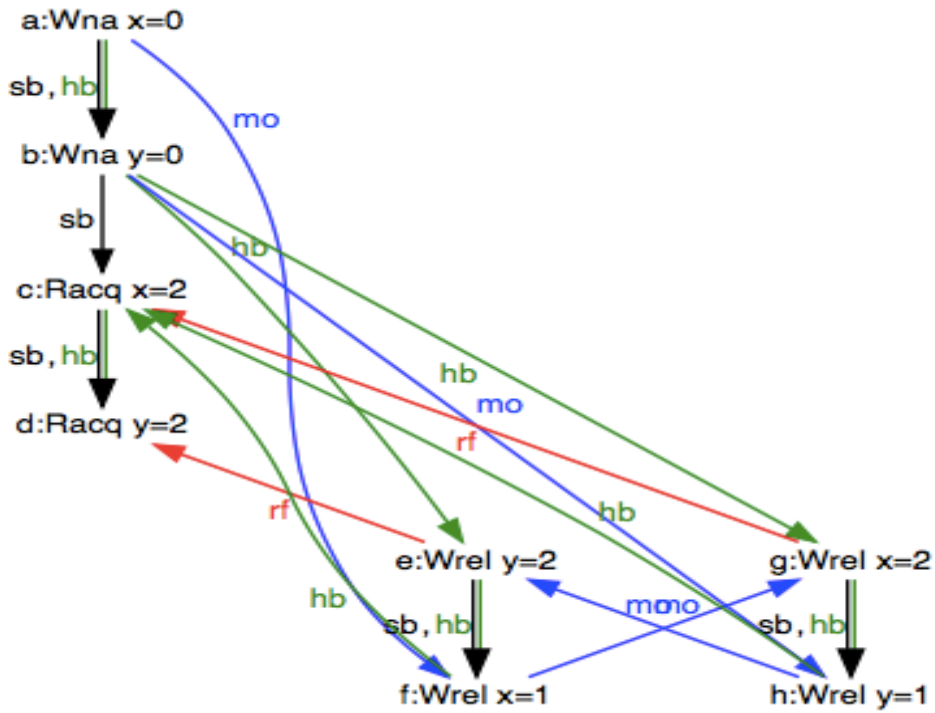
- Can be combinatorially explosive:

  - which values might a read "see"?
  - many potential writes for a given read
  - many possible choices for modification order
  - axioms involve complex interweaving of edge relations
  - expensive checking for cycles

Example

$$(z_1{:=}x_{acq}; z_2{:=}y_{acq}) \| (y_{rel}{:=}2; x_{rel}{:=}1) \| (x_{rel}{:=}2; y_{rel}{:=}1)$$

The execution graph



describes a (non-SC) execution from $[x : 0, y : 0]$ to $[x : 2, y : 2]$

- W_{rel} $x{=}2$, W_{rel} $y{=}2$ are latest in modification order
- W_{rel} $x{=}1$, W_{rel} $y{=}1$ are latest in program order

Analysis Tools

- Obviously we need automated tools
  - to tame the combinatorial explosion
  - to reduce the chance of human error
  - to manage complexity
- Some impressive and useful tools have been built that deal with execution graphs, including:

  `cppmem`, `diy`, `litmus`, ... Alglave, Sewell, et al.
- `cppmem` was used to generate the picture earlier. (Not the error message!)

- But existing tools are only usable on small examples.
- And are execution graphs and axioms the only approach?

  The axiomatic methodology has some inherent limitations...

Issues

- **Not compositional**

  An execution graph describes an entire program.
  No interaction with environment.

- **Complex constraints**

  Must find, or rule out, relations that satisfy axioms.

- **Correctness and completeness**

  How do we know the axioms are "correct" and sufficient?

  Still under investigation...
  Problematic "out-of-thin-air" examples.
  Axiomatics not always true to runtime behavior.
  Recent proposal to modify C11 axioms (Vafeiadis, 2016).

- **Limited applicability**

  Current tools only handle small finite graphs.
  Mainly for partial correctness.

Desiderata

We can benefit from a ***compositional*** semantics

- ***modular***, to help tame complexity
- ***abstract***, to avoid machine-dependence
- ***truly concurrent***, to allow for weak memory

Our Plan

A ***truly concurrent*** denotational semantic framework
for ***weak memory*** with the following characteristics:

(1) Writes to the same variable appear to happen
in the same order, to all threads.

(2) Reads always see the most recently written value[1].

(3) The actions of each thread happen in program order.

(4) Non-atomic code in a race-free program can be optimized,
without affecting execution.

Similar to the characteristics of C11 ***release/acquire****,*
the weak memory model assumed in recent logics

Relaxed Separation Logic *Vafeiadis, Narayan 2013*
GPS *Turon, Dreyer, Vafeiadis 2014*

[1] *modulo* delays attributable to buffering or caching

release/acquire

A write instruction may not write directly to RAM, but write first to a cache. This may cause other threads to "see" the write later, and it can be hard to predict when.

Some processors (e.g. Itanium) offer primitives with acquire/release semantics, and stronger guarantees:

- A read always sees the writes cached by other threads.
- A write is guaranteed to write to RAM (not just to the cache).

In more abstract, less hardware-specific terms:

- An "acquire read" always happens prior to any memory references that occur after it in program order.
- A "release write" always happens after any memory references that occur before it in program order.

Warnings and Promises

- Our approach is *abstract* and independent of hardware
  - no caches or store buffers
- We focus on *foundations* for "true concurrency" semantics
  - generalizing from trace-based semantics
- We discuss *weak memory models* such as
  SC, TSO, release/acquire

  only informally; you should get the main ideas without details.
- Vague, intuitive-sounding terminology, such as
  happens, occurs, sees, perceives
  before, after, prior to, simultaneously

  may be difficult to make precise, and usually isn't.

Rationale

- We will deal with "simple" shared-memory programs:
  - no heap or mutable state
    just shared variables
  - only two kinds of memory access
    atomic, non-atomic
  - one synchronization construct
    mutex locks, or binary semaphores

- Sufficient to introduce main issues and concepts:
  - truly concurrent semantics
  - weak memory phenomena

- Our framework can be adapted and generalized:
  - heap, mutable state
  - other synchronization primitives, e.g. CCR, CAS
  - additional memory access levels, e.g. *sc* ("Java volatile")

Program Syntax

- Abstract grammar

$$i \in \textbf{Ide},\ r \in \textbf{Res}, e \in \textbf{Exp}_{int},\ b \in \textbf{Exp}_{bool},\ c \in \textbf{Com}$$

$$\begin{array}{rcl} e & ::= & n \mid i_\alpha \mid e_1 + e_2 \\ b & ::= & \textbf{true} \mid \neg b \mid b_1 \vee b_2 \mid e_1 = e_2 \mid e_1 < e_2 \\ c & ::= & \textbf{skip} \mid i_\alpha{:=}e \mid c_1; c_2 \mid c_1 \| c_2 \\ & \mid & \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{while } b \textbf{ do } c \\ & \mid & \textbf{lock } r \mid \textbf{unlock } r \mid \textbf{resource } r \textbf{ in } c \\ \alpha & ::= & at \mid na \end{array}$$

- Read and write variable occurrences are tagged α
  - Atomic *at*, non-atomic *na*
  - May omit α when *na*

Program Behavior

Without getting into the details yet...

- Truly concurrent execution of threads

  $i_{at}{:=}v$ behaves like release-write

  $i_{at}{=}v$ behaves like acquire-read

- Atomic accesses don't race

$$x_{at}{:=}1 \parallel x_{at}{:=}2$$

- Non-atomic races get detected

$$x_{at}{:=}1 \parallel y_{na}{:=}x_{na}$$

- Locks are mutually exclusive

$$(\textbf{lock}\ r;\ x_{na}{:=}1; \ldots) \parallel (\textbf{lock}\ r; y_{na}{:=}x_{na}; \ldots)$$

- Blocks are statically scoped

$$(\textbf{resource}\ r\ \textbf{in}\ c_1) \parallel c_2$$

Outline

- **Litmus tests** SC ✓ TSO ✗ rel/acq ✗
  - small programs exhibiting weak memory behavior
  - examples of optimization
- **A denotational framework for weak memory**
  - not global, but *local* state
  - not sequences, but *partial orders*
- **Weak memory *execution*, semantically**
  - litmus tests, revisited
  - optimization theorem
- **Semantic properties**
  - laws of program equivalence
- **Conclusions**
  - advantages and limitations
  - topics for further research

Litmus Test 1

Store Buffering

SC ✗ TSO ✓ rel/acq ✓

$$(x_{at} := 1; z_1 := y_{at}) \parallel (y_{at} := 1; z_2 := x_{at})$$

- Each thread writes one shared variable, then reads the other.
- From initial state

$$[x : 0, y : 0, z_1 : v_1, z_2 : v_2]$$

this program can terminate in

$$[x : 1, y : 1, z_1 : 0, z_2 : 0].$$

Reads may see stale values.

Litmus Test 2

Message Passing

SC ✓ TSO ✓ rel/acq ✓

$$(x{:=}37; y_{at}{:=}1) \parallel (\textbf{while } y_{at} = 0 \textbf{ do skip};\ z{:=}x)$$

- From initial state
$$[x : 0, y : 0, z : v]$$
this program is race-free,
even though the accesses to x are not atomic,
and ends with $z = 37$.
- The while-loop only terminates after the write to y.

If a thread sees a write, it sees everything that happened before that write.

Litmus Test 3

Independent Reads of Independent Writes

SC ✗ TSO ✗ rel/acq ✓

$$x_{at}{:=}1 \parallel y_{at}{:=}1 \parallel (z_1{:=}x_{at};\ z_2{:=}y_{at}) \parallel (w_1{:=}y_{at};\ w_2{:=}x_{at})$$

- From an initial state with

$$x = y = 0$$

this program can terminate with

$$z_1 = w_1 = 1,\ z_2 = w_2 = 0.$$

- In this execution
  one thread sees the write to x before the write to y,
  one thread sees the write to y before the write to x.

No total ordering on writes to different variables.

Litmus Test 4

Coherence

SC ✓ TSO ✓ rel/acq ✓

$$x_{at}{:=}1 \parallel x_{at}{:=}2 \parallel (z_1{:=}x_{at};\ z_2{:=}x_{at}) \parallel (w_1{:=}x_{at};\ w_2{:=}x_{at})$$

- When started with all variables equal to 0,

  $$x = z_1 = z_2 = w_1 = w_2 = 0$$

  this program has no execution that ends with

  $$z_1 = w_2 = 1,\ z_2 = w_1 = 2.$$

- Both threads see the writes to x in the same order.

Total ordering on the writes to a single variable.

Not true in some even weaker memory models,
but desirable for effective reasoning, as in GPS, RSL.

Litmus Test 5

Optimization

SC ✓ TSO ✓ rel/acq ✓

- If c uses r to access x,

$$\begin{aligned} & c \parallel \textbf{lock}\ r; x{:=}x+1; x{:=}x+1; \textbf{unlock}\ r \\ \equiv\ & c \parallel \textbf{lock}\ r; x{:=}x+2; \textbf{unlock}\ r \end{aligned}$$

- Non-atomic writes to different variables can be re-ordered

$$\begin{aligned} & c \parallel (c_{11}; x{:=}1; y{:=}2; c_{12}) \\ \equiv\ & c \parallel (c_{21}; y{:=}2; x{:=}1; c_{22}) \end{aligned}$$

Here, informally, $\equiv$ means "has same execution results".

In a race-free program, non-atomic code can be optimized without affecting execution.

Taking stock

- Traditional semantics only works for SC.
- Modern architectures don't behave like SC.
- Instead they provide weak or relaxed memory access.
- Litmus tests exhibit characteristic weak memory "features":
  - stale reads
  - no total order on all writes
  - total order per single variable

- We need *good* semantics for "truly concurrent" programs
  - suitable for exploring the weak memory spectrum

A denotational framework for weak memory

$$\text{framework} = \text{semantics} + \text{execution}$$

- **An abstract *denotational semantics***
  - states and actions
  - footprints and effects
  - partially ordered multisets
- **A semantically-based definition of *execution***
  - exhibiting weak memory behaviors
- **A framework for exploration**
  - alternative forms of execution
  - embodying other weak memory models
  - classification and clarification

States

$$\sigma, \tau \in \Sigma = \mathbf{Ide} \rightharpoonup_{fin} V_{int}$$

- A state is a finite partial function from identifiers to values.

$$[x_1 : v_1, \ldots, x_n : v_n] \qquad \{(x_i, v_i) \mid 1 \leq i \leq n\}$$

- We write $[\sigma \mid \tau]$ for the state obtained by updating σ with τ

$$[\sigma \mid \tau] = (\sigma \backslash dom\, \tau) \cup \tau$$

$$\begin{array}{rcll} [\sigma \mid \tau](i) & = & \tau(i) & \text{if } i \in dom(\tau) \\ & = & \sigma(i) & \text{if } i \in dom(\sigma) - dom(\tau) \end{array}$$

- States σ_1 and σ_2 are *consistent*, written as $\sigma_1 \Uparrow \sigma_2$, iff they *agree* on $dom(\sigma_1) \cap dom(\sigma_2)$, i.e.

$$\forall i \in dom(\sigma_1) \cap dom(\sigma_2).\ \sigma_1(i) = \sigma_2(i).$$

When this happens, we have

$$[\sigma_1 \mid \sigma_2] = [\sigma_2 \mid \sigma_1] = \sigma_1 \cup \sigma_2 \in \Sigma.$$

Actions

$$\lambda, \mu \in \Lambda$$

$$\begin{array}{lcl} \lambda & ::= & \delta \mid i_\alpha{=}v \mid i_\alpha{:=}v \mid \mathit{lock}\ r \mid \mathit{unlock}\ r \\ \alpha & ::= & \mathit{at} \mid \mathit{na} \end{array}$$

- δ is an idle action.
- Reads $i_\alpha{=}v$ and writes $i_\alpha{:=}v$
  are tagged as atomic *at* or non-atomic *na*.
  We may omit α when non-atomic.
- The *lock* and *unlock* actions are atomic.

Traces

- A trace is a finite or infinite sequence of actions

$$\alpha, \beta \in \Lambda^{\infty} = \Lambda^{*} \cup \Lambda^{\omega}$$

- We write $\alpha\beta$ for the trace obtained by concatenating β onto α

$$\alpha\beta = \alpha \quad \text{if } \alpha \text{ infinite}$$

- Examples

$$x_{at}{:=}1\ x_{at}{=}1\ y{:=}1$$
$$x_{at}{:=}1\ x_{at}{=}0\ y{:=}0$$

consecutive reads and writes
need not be sequentially executable

- A trace is *sequential* iff its actions are sequentially executable

Footprints

Definition

The footprint of an action

$$[\![\lambda]\!] \subseteq \Sigma \times \Sigma^{\top}$$

is given by

$$\begin{aligned}
[\![\delta]\!] &= \{([\,], [\,])\} \\
[\![i_\alpha{:=}v]\!] &= \{([i : v], [\,])\} \\
[\![i_\alpha{:=}v]\!] &= \{([i : v_0], [i : v]) \mid v_0 \in V_{int}\} \\
[\![\mathit{lock}\ r]\!] &= \{([r : 0], [r : 1])\} \\
[\![\mathit{unlock}\ r]\!] &= \{([r : 1], [r : 0])\}
\end{aligned}$$

- Describes minimal state needed by, and affected by, an action.
- λ is *enabled in* σ iff there is a $(\sigma_1, \tau_1) \in [\![\lambda]\!]$ such that $\sigma \supseteq \sigma_1$.
- When $(\sigma_1, \tau_1) \in [\![\lambda]\!]$ we say that λ *reads* σ_1, *writes* τ_1.

Effects

Definition

The *effect* of an action

$$\mathcal{E}(\lambda) \subseteq \Sigma \times \Sigma^{\top}$$

is given by

$$\mathcal{E}(\lambda) = \{(\sigma, [\sigma \mid \tau_1]) \mid \exists \sigma_1 \subseteq \sigma.\ (\sigma_1, \tau_1) \in [\![\lambda]\!]\}$$

Theorem

Actions have the following effects:

$$\begin{aligned}
&\mathcal{E}(\delta) = \{(\sigma, \sigma) \mid \sigma \in \Sigma\} \\
&\mathcal{E}(i_\alpha{=}v) = \{(\sigma, \sigma) \mid i : v \in \sigma\} \\
&\mathcal{E}(i_\alpha{:=}v) = \{(\sigma, [\sigma \mid i : v]) \mid i \in \mathit{dom}(\sigma)\} \\
&\mathcal{E}(\mathit{lock}\ r) = \{(\sigma, [\sigma \mid r : 1]) \mid r : 0 \in \sigma\} \\
&\mathcal{E}(\mathit{unlock}\ r) = \{(\sigma, [\sigma \mid r : 0]) \mid r : 1 \in \sigma\}
\end{aligned}$$

Sequential execution

For actions and traces, we define:

$$\sigma \stackrel{\lambda}{\Longrightarrow} \sigma' \quad \text{iff} \quad (\sigma, \sigma') \in \mathcal{E}(\lambda)$$

$$\sigma \stackrel{\lambda_1 \cdots \lambda_n}{\Longrightarrow} \sigma' \quad \text{iff} \quad \sigma \stackrel{\lambda_1}{\Longrightarrow} \cdots \stackrel{\lambda_n}{\Longrightarrow} \sigma'$$

Basic facts:

$$\begin{array}{rlll}
\sigma \stackrel{\delta}{\Longrightarrow} \sigma & \text{always} & & \\
\sigma \stackrel{i_\alpha = v}{\Longrightarrow} \sigma & \text{iff} & (i, v) \in \sigma & \\
\sigma \stackrel{i_\alpha := v}{\Longrightarrow} \sigma' & \text{iff} & i \in dom(\sigma) \;\&\; \sigma' = [\sigma \mid i : v] & \\
\sigma \stackrel{lock\ r}{\Longrightarrow} \sigma' & \text{iff} & \sigma(r) = 0 \;\&\; \sigma' = [\sigma \mid r : 1] & \\
\sigma \stackrel{unlock\ r}{\Longrightarrow} \sigma' & \text{iff} & \sigma(r) = 1 \;\&\; \sigma' = [\sigma \mid r : 0] &
\end{array}$$

We say $\alpha \in \Lambda^*$ is (sequentially) executable from σ if $\exists \sigma'.\ \sigma \stackrel{\alpha}{\Longrightarrow} \sigma'$

Checkpoint

- We introduced traces, states and effects.
- The chosen basis for traditional SC denotational semantics.
- But now let's reconsider and reflect ...

True Concurrency

- Traces were used to model SC concurrency.
  - A trace is a ***linearly ordered*** (multi-)set of actions.
  - Parallel composition = non-deterministic interleaving.
- That's ***false concurrency***. It's not always reasonable to conflate concurrency with non-determinism.

- To handle weaker memory models, and obtain a more philosophically defensible semantics, we must embrace ***true concurrency***

  ... by abandoning ***linearity***.
- This leads to a natural generalization of traces:
  - A pomset is a ***partially ordered*** (multi-)set of actions.

Action Pomsets

cf. Pratt 1986 [2]

Definition

An action pomset $(P, <)$ is a multiset P of *actions*,
with a partial order $<$ on P such that

(a) $<$ is irreflexive, transitive, cycle-free

(b) $<$ has *locally finite height*:
For every $\lambda \in P$ there are finitely many $\mu \in P$ such that $\mu < \lambda$.

We use $<$ to represent a "program order"
on the set P of atomic actions of a program.

Let **Pom** be the set of action pomsets.

[2] Pratt does not require (b).

Pomset Representation

A pomset $(P, <)$ can be seen as a directed acyclic graph

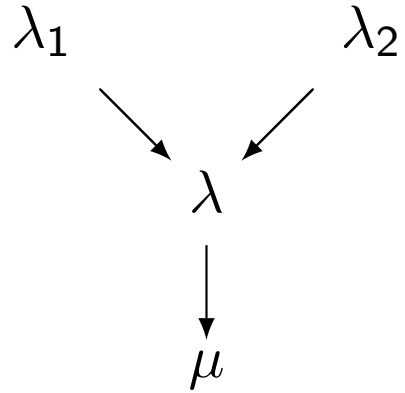
$$G = (V, E, \Phi)$$

with nodes V, edges E, and labeling function $\Phi : V \to \Lambda$.

- "occurrence of λ in P" = "node labelled λ in G"
- "$\lambda < \mu$ in P" = "nodes a, b labelled λ, μ such that $(a, b) \in E^*$

Example

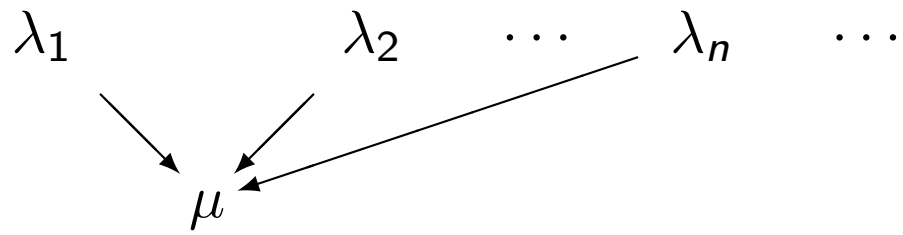
$$V = \{a, b, c, d\}$$
$$E = \{(a, c), (b, c), (c, d)\}$$
$$\Phi = \{(a, \lambda_1), (b, \lambda_2), (c, \lambda), (d, \mu)\})$$



Transitivity edges omitted...

Locally Finite Height

Not allowed



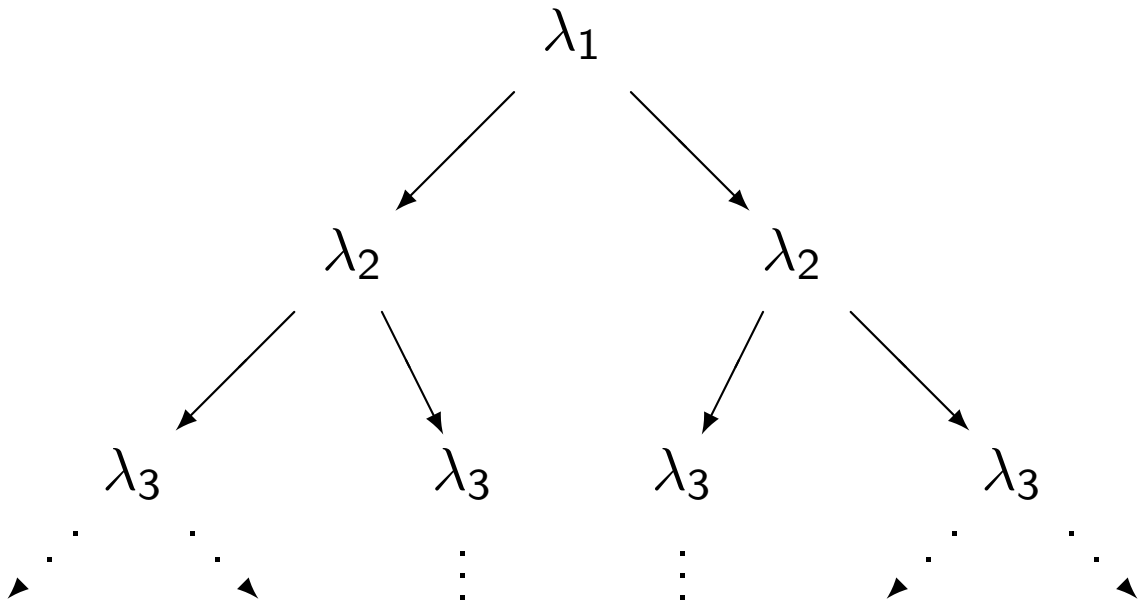
Not allowed

$$\lambda_1 \to \lambda_2 \to \cdots \to \lambda_n \to \cdots \longrightarrow \mu$$

μ not finitely reachable

Locally Finite Height

Allowed



Infinite total height, but each λ_n is finitely reachable.

Linear

Definition

$(P, <)$ is *linear* if

$$\forall \lambda, \mu \in P.\ (\lambda \leq \mu \text{ or } \mu \leq \lambda).$$

- A linear pomset is (isomorphic to) a trace.
- May use trace-like notation, e.g.

$$x{:=}1 \qquad x{:=}1\, y{:=}1\, x{:=}2$$

$$\downarrow$$

$$y{:=}1$$

$$\downarrow$$

$$x{:=}2$$

Linear for r

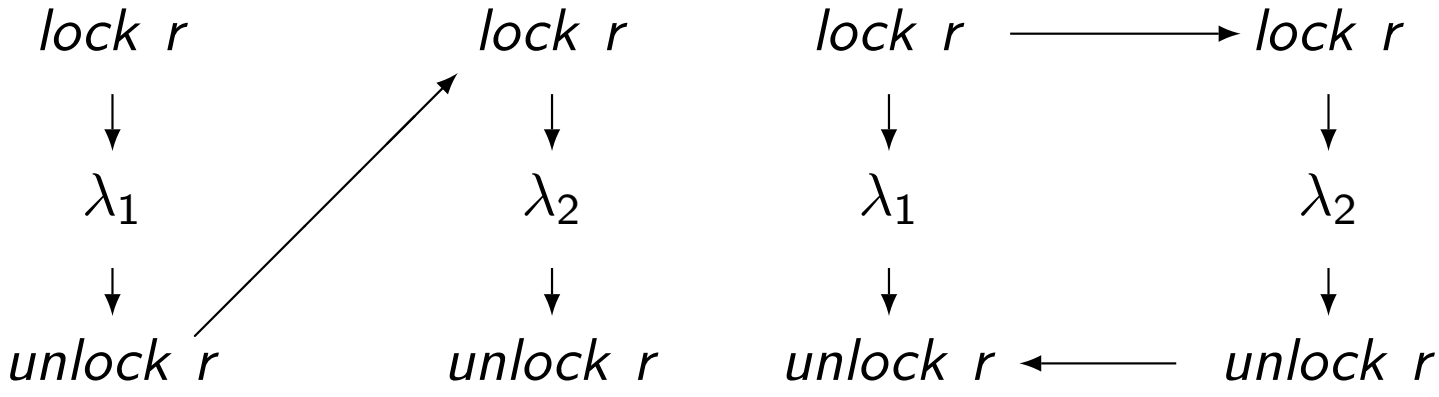
Definition

$(P, <)$ is *linear for* r if

$$\forall \lambda, \mu \in P \upharpoonright r.\ \lambda \leq \mu \text{ or } \mu \leq \lambda.$$

- $P \upharpoonright r$ is the restriction of $(P, <)$ to the actions on r.
- When P is linear for r, $P \upharpoonright r$ is (isomorphic to) a trace.

Examples



Building Pomsets

cf. Pratt 1986 [3]

Parallel composition

- Actions of P_1 and P_2 are independent

$$(P_1, <_1) \parallel (P_2, <_2) \;=\; (P_1 \uplus P_2, <_1 \uplus <_2)$$

Sequential composition

- Actions of P_1 before actions of P_2

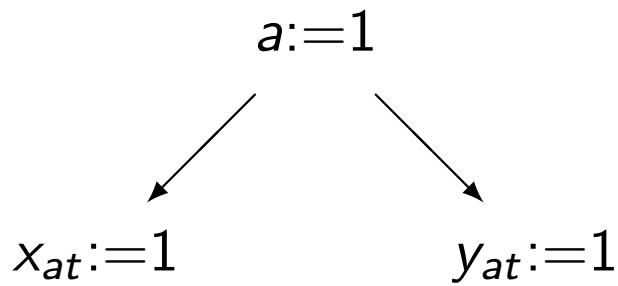
$$\begin{aligned}(P_1, <_1); (P_2, <_2) \;&=\; (P_1, <_1) \\ &\qquad \text{if } \; |P_1| \text{ is infinite} \\ &=\; (P_1 \uplus P_2,\; <_1 \uplus <_2 \uplus P_1 \times P_2) \\ &\qquad \text{if } \; |P_1| \text{ is finite}\end{aligned}$$

These operations extend pointwise to sets of pomsets.

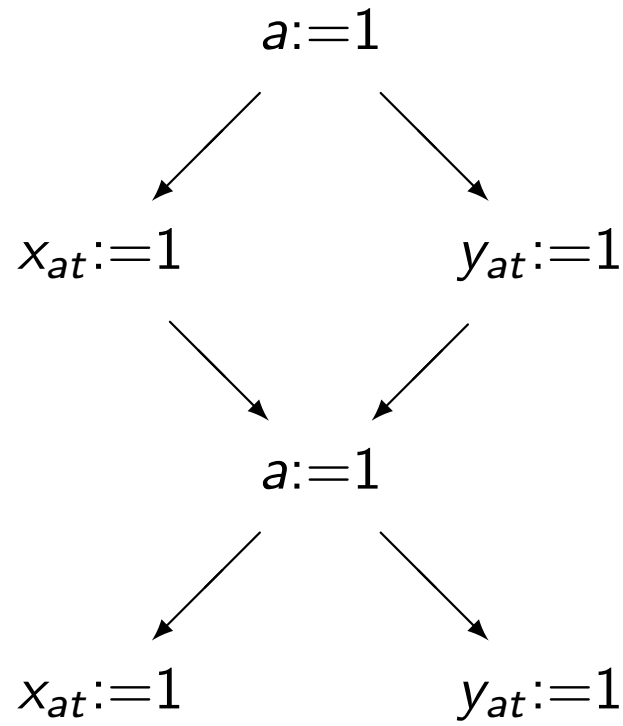
[3] Pratt's $P_1; P_2$ does not handle the infinite case separately.

Sequential composition

P

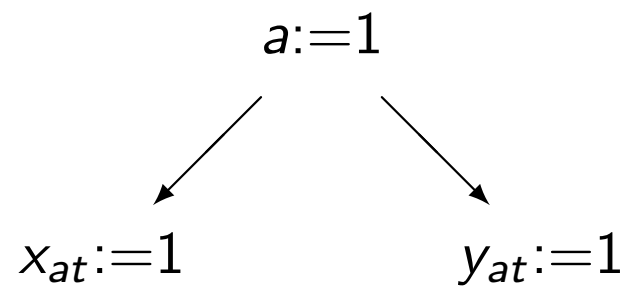


$P; P$

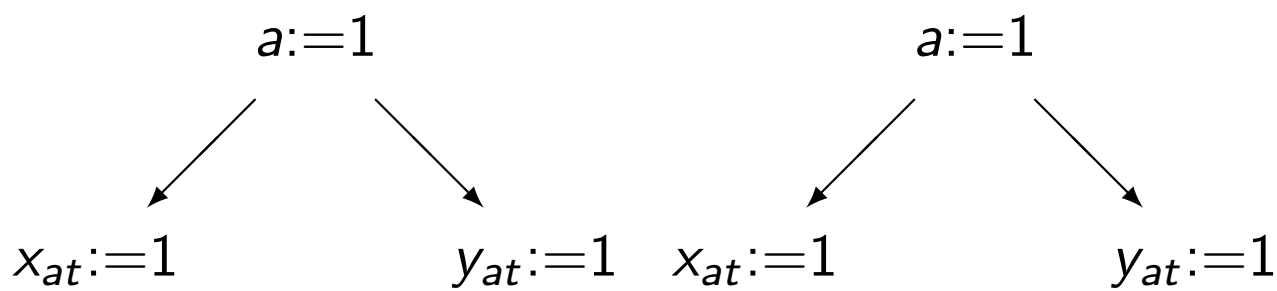


Parallel composition

P



$P \| P$



Pomset Properties

$$
\begin{array}{rcl}
P_1;(P_2;P_3) & = & (P_1;P_2);P_3 \\
P_1 \| (P_2 \| P_3) & = & (P_1 \| P_2) \| P_3 \\
\\
P_1 \| P_2 & = & P_2 \| P_1 \\
\\
P \| P & \neq & P \\
\\
(P_1 \| P_2);P_3 & \neq & (P_1;P_3) \| (P_2;P_3) \\
P_1;(P_2 \| P_3) & \neq & (P_1;P_2) \| (P_1;P_3)
\end{array}
$$

- $P = Q$ means "up to isomorphism"
- Same partial order, same multiplicities for each $\lambda \in \Lambda$

Pomset Iteration

Definition

- For $n \geq 0$ let P^n be the n-fold sequential composition.

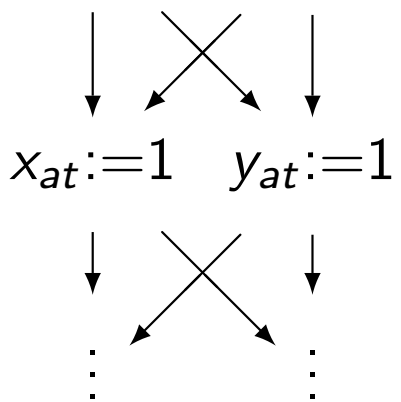
$$\begin{aligned} P^0 &= \{\delta\} \\ P^{k+1} &= P ; P^k \end{aligned}$$

- Let P^ω be the countably infinite sequential composition.
- Let $P^* = \bigcup_{n=0}^{\infty} P^n$.

Example

P : $x_{at}{:=}1$ $y_{at}{:=}1$

P^ω : $x_{at}{:=}1$ $y_{at}{:=}1$



Pomset Extension

Definition

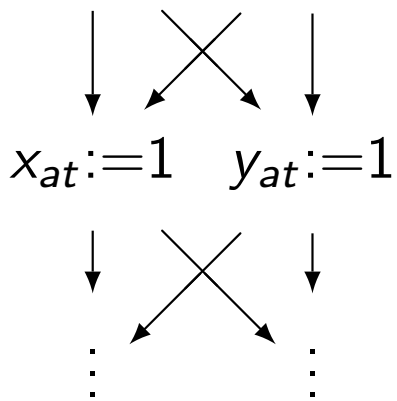
- $(P', <')$ *extends* $(P, <)$ if $P \subseteq P'$ and $< \subseteq <'$

Example

- P^ω extends P

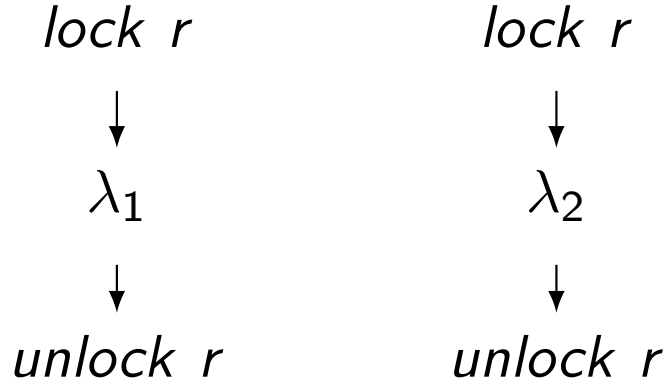
P : $x_{at}{:=}1$ $y_{at}{:=}1$

P^ω : $x_{at}{:=}1$ $y_{at}{:=}1$



Exercise

- Find all pomset extensions of



  that are linear for r.
- Which of these are sequentially executable from $[r : 0]$?

  (Assume that the actions λ_1 and λ_2 do not involve r.)

Pomset operations

- Let $\mathbb{P}(\mathbf{Pom})$ be the powerset of **Pom**.

  Programs denote sets of pomsets.

- $\mathbb{P}(\mathbf{Pom})$ is a complete lattice, ordered by set inclusion.
- Pomset operations extend to sets of pomsets:

  For $X, Y \subseteq \mathbf{Pom}$ we write

$$\begin{aligned} X; Y &= \{P; Q \mid P \in X, Q \in Y\} \\ X \| Y &= \{P \| Q \mid P \in X, Q \in Y\} \end{aligned}$$

- Similarly for iteration:

$$\begin{aligned} X^n &= \{P_1; \cdots; P_n \mid P_i \in X\} \\ X^\omega &= \{P_1; \cdots; P_n; \cdots \mid P_i \in X\} \end{aligned}$$

  but note that X^n is not the same as $\{P^n \mid P \in X\}$.

- These operations are monotone and continuous.

Pomset Semantics

for expressions

$$\mathcal{P} : \mathbf{Exp}_{int} \to \mathbb{P}(\mathbf{Pom} \times V_{int})$$
$$\mathcal{P} : \mathbf{Exp}_{bool} \to \mathbb{P}(\mathbf{Pom} \times V_{bool})$$

are defined by structural induction, e.g.

$$\mathcal{P}(i_\alpha) = \{(\{i_\alpha{=}v\}, v) \mid v \in V_{int}\}$$
$$\mathcal{P}(e_1 + e_2) = \{(P_1 \| P_2, v_1 + v_2) \mid (P_1, v_1) \in \mathcal{P}(e_1),\ (P_2, v_2) \in \mathcal{P}(e_2)\}$$

$$\mathcal{P}(e_1 = e_2) = \{(P_1 \| P_2, v_1 = v_2) \mid (P_1, v_1) \in \mathcal{P}(e_1),\ (P_2, v_2) \in \mathcal{P}(e_2)\}$$

Let $\mathcal{P}(b)_{tt}, \mathcal{P}(b)_{ff} \in \mathbb{P}(\mathbf{Pom})$ be

$$\mathcal{P}(b)_{tt} = \{P \mid (P, tt) \in \mathcal{P}(b)\}$$
$$\mathcal{P}(b)_{ff} = \{P \mid (P, ff) \in \mathcal{P}(b)\}$$

Examples

- The expression $x_{at} + x_{at}$ has a pomset entry of form

$$(\{x_{at}{=}v_1\} \| \{x_{at}{=}v_2\}, v_1 + v_2),$$

  for all $v_1, v_2 \in V_{int}$.

- The boolean expression $x = x$ has

$$\mathcal{P}(x = x)_{\mathtt{tt}} = \{\{x{=}v_1\} \| \{x{=}v_2\} \mid v_1 = v_2\}$$
$$\mathcal{P}(x = x)_{\mathtt{ff}} = \{\{x{=}v_1\} \| \{x{=}v_2\} \mid v_1 \neq v_2\}$$

We allow for interaction with environment.

Pomset Semantics

for commands

$$\mathcal{P} : \mathbf{Com} \to \mathbb{P}(\mathbf{Pom})$$

is defined by structural induction:

$$\begin{aligned}
&\mathcal{P}(\mathbf{skip}) = \{\{\delta\}\} \\
&\mathcal{P}(i_\alpha{:=}e) = \{P; \{i_\alpha{:=}v\} \mid (P, v) \in \mathcal{P}(e)\} \\
&\mathcal{P}(c_1; c_2) = \mathcal{P}(c_1); \mathcal{P}(c_2) \\
&\mathcal{P}(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) = \mathcal{P}(b)_{tt}; \mathcal{P}(c_1)\ \cup\ \mathcal{P}(b)_{ff}; \mathcal{P}(c_2) \\
&\mathcal{P}(\mathbf{while}\ b\ \mathbf{do}\ c) = (\mathcal{P}(b)_{tt}; \mathcal{P}(c))^*; \mathcal{P}(b)_{ff}\ \cup\ (\mathcal{P}(b)_{tt}; \mathcal{P}(c))^\omega \\
&\mathcal{P}(c_1 \| c_2) = \mathcal{P}(c_1) \parallel \mathcal{P}(c_2) \\
&\mathcal{P}(\mathbf{lock}\ r) = \{\{lock\ r\}\} \\
&\mathcal{P}(\mathbf{unlock}\ r) = \{\{unlock\ r\}\} \\
&\mathcal{P}(\mathbf{resource}\ r\ \mathbf{in}\ c) = \{P \backslash r \mid P \in \mathcal{P}(c)_r\}
\end{aligned}$$

Example

The program

$$y_{at}{:=}1 \parallel \textbf{if } y_{at}{=}0 \textbf{ then } x{:=}y_{at}+1 \textbf{ else skip}$$

denotes the set of pomsets

$$\{P_v \mid v \in V_{int}\} \cup \{Q_v \mid v \neq 0\},$$

where

$$P_v: \; y_{at}{:=}1 \quad y_{at}{=}0 \;\downarrow\; y_{at}{=}v \;\downarrow\; x{:=}v+1 \qquad Q_v: \; y_{at}{:=}1 \quad y_{at}{=}v \;\downarrow\; \delta$$

Example

The program

$$y_{at}{:=}1 \parallel \textbf{while } y_{at}{=}0 \textbf{ do } x_{at}{:=}x_{at}+1$$

has pomsets of form

$$y_{at}{:=}1 \parallel P$$

where $P \in loop^{*}stop \cup loop^{\omega}$,
and

$$\begin{array}{lcl} loop & = & \{\{y_{at}{=}0\, x_{at}{=}v\, x_{at}{:=}v+1\} \mid v \in V_{int}\} \\ stop & = & \{\{y_{at}{=}v\} \mid v \neq 0\} \end{array}$$

Draw some of these and understand what kinds of interactive behavior they represent.

What is special about the pomsets in $y_{at}{:=}1 \parallel loop^{\omega}$?

Local Resources

in more detail

$$\mathcal{P}(\textbf{resource } r \textbf{ in } c) = \{P\backslash r \mid \ P \in \mathcal{P}(c)_r\}$$

(i) $\mathcal{P}(c)_r$ is the set of all pomsets $(P, <')$ constructible by picking a $(P, <) \in \mathcal{P}(c)$ and linearizing the r-actions so that $P \upharpoonright r$ is *sequentially executable* from $[r : 0]$

(ii) $P\backslash r$ *erases* the r-actions from $(P, <')$.

Intuition

The local resource r is initially "available" $(r = 0)$, and not accessible by other threads:

(i) r-actions of c get executed sequentially from $[r : 0]$, without interference by other threads:

$$\mathit{lock}\ r \cdots \mathit{unlock}\ r \cdots \mathit{lock}\ r \cdots$$

(ii) r-actions are invisible outside the scope

Only uses the pomsets of c that can be suitably extended

Pomset Semantics

Nature and Purpose

- The pomset semantics of programs is defined without dependence on memory model or machine architecture
  - abstract, high-level
  - no need for weak memory axioms
  - no need to pick a specific memory model
  - denotational, so designed to be compositional
- We kept actions *distinct* from effects
  - pomset structure shows *program order*
  - no need to track the *state* (yet!)
- Can serve as a *tabula rasa*

  . . . like a sheet of paper ready for writing upon.

Using Pomset Semantics

To illustrate how the semantic clauses work. . .

Recall the litmus test programs

store buffering, message-passing, IRIW, . . .

We now examine their pomset semantics.

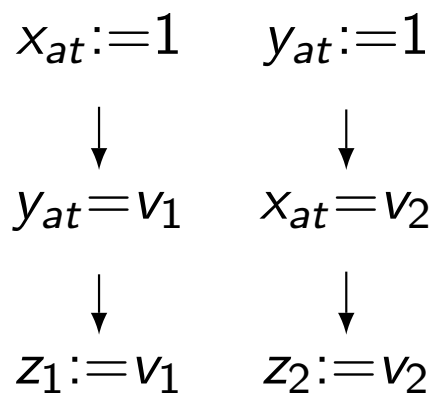
In each case the pomsets of the program
capture its essential computational structure,
and ignore irrelevant aspects.

Later we will explore executional behavior (and track the state).

Store Buffering

$$(x_{at} := 1; z_1 := y_{at}) \parallel (y_{at} := 1; z_2 := x_{at})$$

- Each pomset of this program has the form



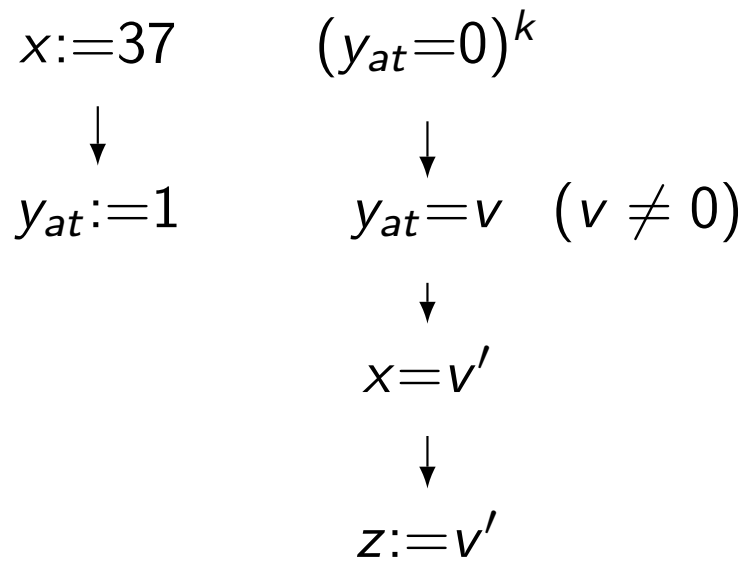
  where $v_1, v_2 \in V_{int}$.
- We include "non-sequential" cases like $v_1 = 42, v_2 = 63$, to allow for behavior in parallel contexts, e.g.

$$- \parallel x_{at} := 63 \parallel y_{at} := 42$$

Message Passing

$$(x{:=}37; y_{at}{:=}1) \parallel (\textbf{while } y_{at} = 0 \textbf{ do skip}; \ z{:=}x)$$

- Each finite pomset of this program has the form

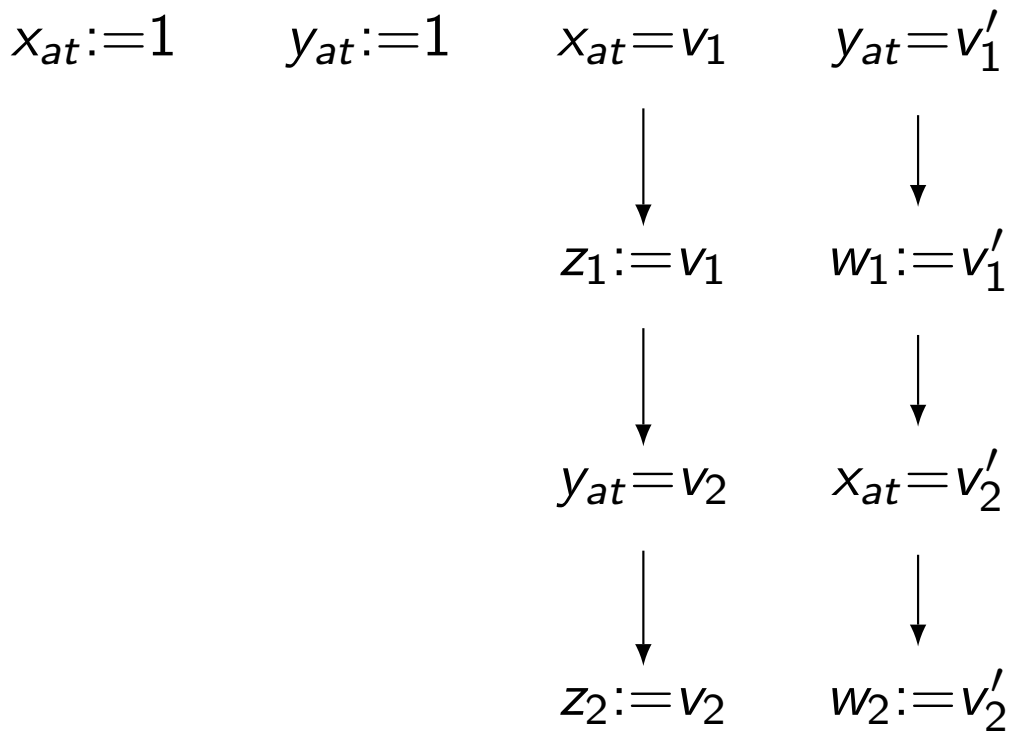


  with $k \geq 0$ and $v, v' \in V_{int}$
- Also has the infinite pomset $\{x{:=}37\, y_{at}{:=}1,\ (y_{at}{=}0)^\omega\}$

Independent Reads of Independent Writes

$$x_{at}{:=}1 \parallel y_{at}{:=}1 \parallel (z_1{:=}x_{at};\ z_2{:=}y_{at}) \parallel (w_1{:=}y_{at};\ w_2{:=}x_{at})$$

has pomsets of form

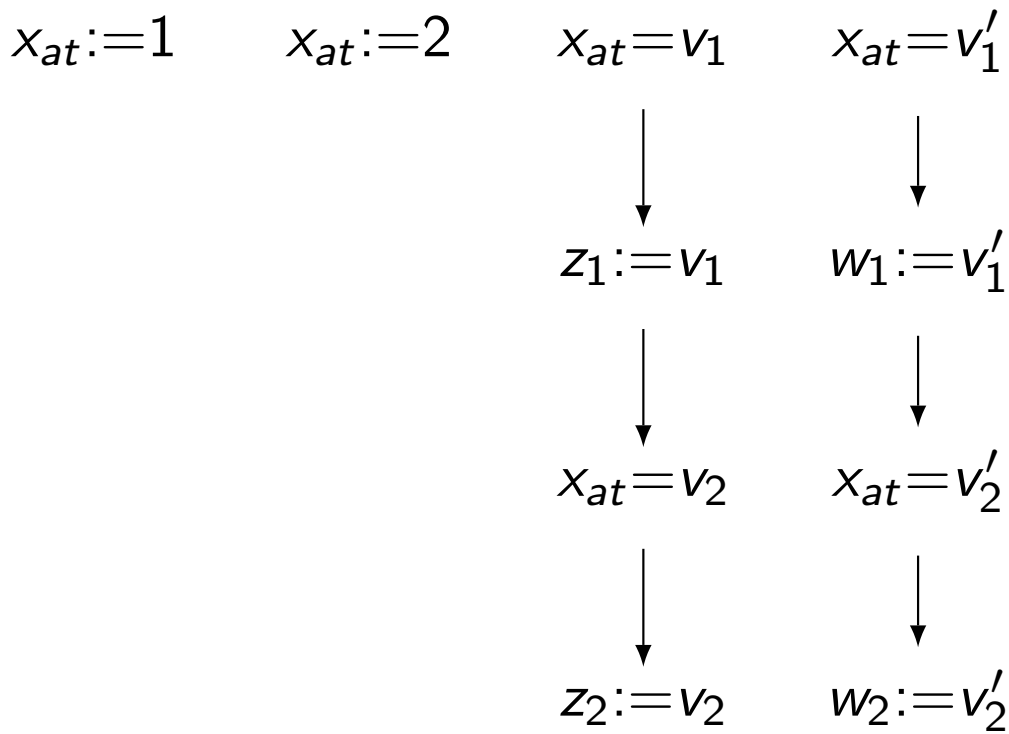


for $v_1, v_2, v_1', v_2' \in V_{int}$

Coherence

$$x_{at}{:=}1 \parallel x_{at}{:=}2 \parallel (z_1{:=}x_{at};\ z_2{:=}x_{at}) \parallel (w_1{:=}x_{at};\ w_2{:=}x_{at})$$

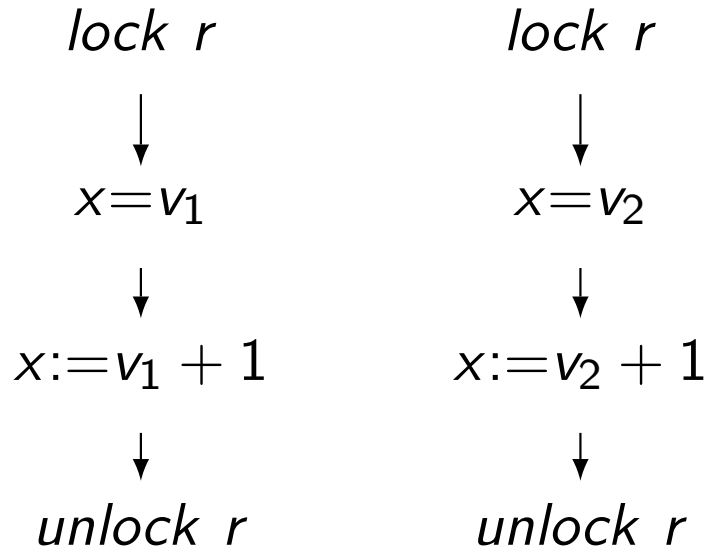
has pomsets of form



for $v_1, v_2, v_1', v_2' \in V_{int}$

Concurrent Increments

- Let *inc* be **lock** r; $x{:=}x+1$; **unlock** r.
- Pomsets for $inc \parallel inc$ have form

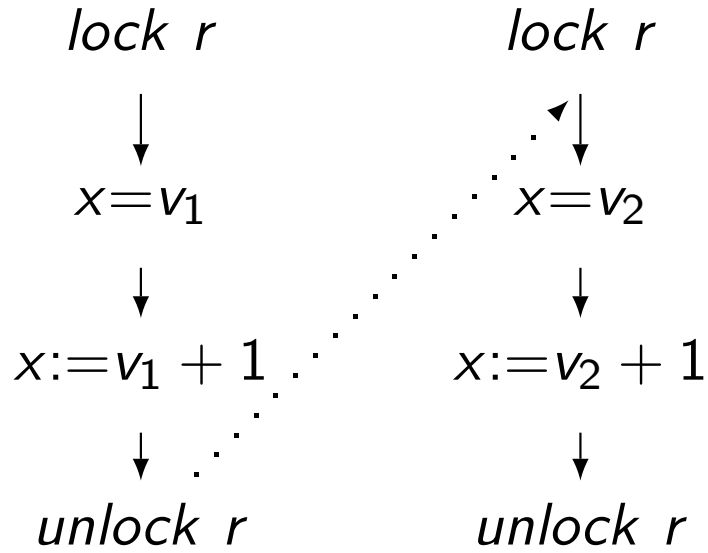


for $v_1, v_2 \in V_{int}$.

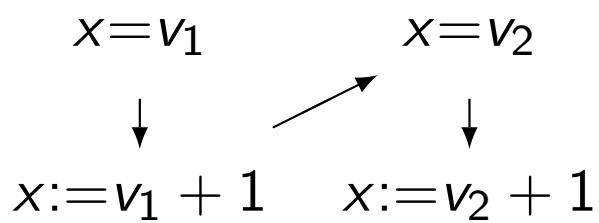
Concurrent Increments using a private lock

resource r **in** $(inc \parallel inc)$

- The (relevant) pomsets in $\mathcal{P}(inc \| inc)_r$ have the form



and after erasing the r-actions, we get



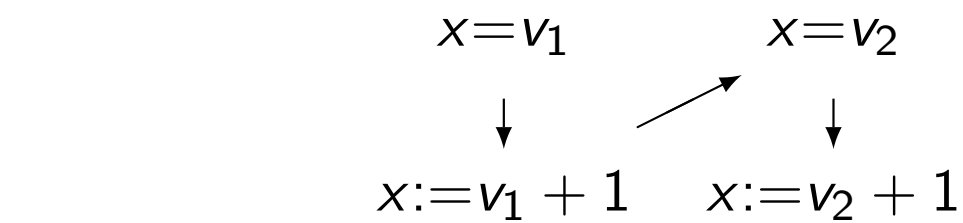
Semantic equivalence

$$\mathcal{P}(\textbf{resource } r \textbf{ in } (inc \| inc)) = \mathcal{P}(x{:=}x+1; x{:=}x+1)$$

- The pomsets of

  $$\textbf{resource } r \textbf{ in } (inc \| inc)$$

  are

  with $v_1, v_2 \in V_{int}$
- These are also the pomsets of

  $$x{:=}x+1; x{:=}x+1$$

  Accesses to x are non-atomic, but get serialized

  Use of the private lock ensures race-freedom

A Semantic Framework

$$\textit{framework} = \textit{semantics} + \textit{execution}$$

- We have a denotational semantics

$$\mathcal{P} : \mathbf{Com} \to \mathbb{P}(\mathbf{Pom})$$

  defined without reference to memory model or architecture.
- We can now introduce a notion of

$$\textit{pomset execution}$$

  for this semantics, tailored to reflect the assumptions and guarantees of a particular memory model, or a family of memory models sharing certain characteristics.
- We will do this next for one such memory model, characterized implicitly.

Pomset Execution

- We will define a form of *pomset execution* tailored for a weak memory model in which
  (1) Writes to the same variable appear in same order to all threads.
  (2) Reads see the most recently written value.
  (3) The actions of each thread happen in program order.
  (4) In a race-free program, non-atomic code can be optimized.

- The key is to extend footprints (and effects) to pomsets.
  - Action footprints may be composable, in sequence or in parallel
  - Must take account of state, program order, atomicity

- We need first to identify when pomsets P_1 and P_2 are
  - *consecutively* executable, or
  - *concurrently* executable

  components of a pomset P.

Sequencing

Initial Segment

- $P_1 \subseteq P$ is an *initial segment* of $(P, <)$ if P_1 is down-closed:

$$\forall \lambda \in P_1, \mu \in P.\ \mu < \lambda \ \Rightarrow\ \mu \in P_1.$$

- $P = P_1 \triangleleft P_2$ when P_1 is an initial segment and P_2 is the rest.

Properties

- $(P_1 \triangleleft P_2) \triangleleft P_3 = P_1 \triangleleft (P_2 \triangleleft P_3)$.
- When $P = P_1 \triangleleft P_2$ and $Q = Q_1 \triangleleft Q_2$, it follows that

$$P \uplus Q = (P_1 \uplus Q_1) \triangleleft (P_2 \uplus Q_2).$$

Examples

For linear pomsets, i.e. traces,

$$\begin{array}{rcl} \text{initial segment} & = & \text{prefix} \\ \lhd & = & \text{concatenation} \end{array}$$

- If P is linear and $P = P_1 \lhd P_2$,
  then P_1 and P_2 are linear and $P = P_1; P_2$.
- The converse also holds.

For non-linear pomsets:

- When $P = P_1 \| P_2$ we get

$$P = P_1 \lhd P_2 \text{ and } P = P_2 \lhd P_1$$

but $P \neq P_1; P_2$ and $P \neq P_2; P_1$.

Concurrence

- We write P_1 **co** P_2 to mean that there is ***no race condition***: no variable is written by one pomset and read or written *non-atomically* by the other.
  - Atomic writes to the same variable are allowed.
  - When all actions of P_1 and P_2 are non-atomic they must write to disjoint sets of variables.

- We say that P_1 and P_2 are ***concurrent***, $P_1 \perp P_2$, iff they do not race, and use disjoint locks:
  - $P_1 \perp P_2$ iff P_1 **co** P_2 & $res(P_1) \cap res(P_2) = \{\}$.
  - $res(P)$ is the set of lock names r in actions of P.

Pomset Footprints

Intuition

$$[\![P]\!] \subseteq \Sigma \times \Sigma^{\top}$$

$[\![P]\!]$ contains footprint pairs for all ways to execute the actions in P while respecting the "program order" $<$ and the following rules:

- An initial action can be done if enabled (Act)
- Consecutive initial segments can be done in sequence (Seq)
- Concurrent initial segments can be done in parallel (Par) and we detect race conditions (Race)
- Writes to the same variable are linearly ordered

Pomset Footprints

Definition

The pomset footprint function

$$[\![-]\!] : \mathbf{Pom} \to \mathbb{P}(\Sigma \times \Sigma^{\top})$$

is the least function such that:

ACT If P is a singleton $\{\lambda\}$, $[\![P]\!] = [\![\lambda]\!]$.

SEQ If $P = P_1 \triangleleft P_2$, $(\sigma_1, \tau_1) \in [\![P_1]\!]$, $(\sigma_2, \tau_2) \in [\![P_2]\!]$, $[\sigma_1 \mid \tau_1] \Uparrow \sigma_2$, then $(\sigma_1 \cup (\sigma_2 \backslash dom\, \tau_1), [\tau_1 \mid \tau_2]) \in [\![P]\!]$.
If $P = P_1 \triangleleft P_2$ and $(\sigma, \top) \in [\![P_1]\!]$, then $(\sigma, \top) \in [\![P]\!]$.

PAR If $P = P_1 \uplus P_2$, P_1 **co** P_2, $res(P_1) \cap res(P_2) = \{\}$, $(\sigma_1, \tau_1) \in [\![P_1]\!]$, $(\sigma_2, \tau_2) \in [\![P_2]\!]$, and $\sigma_1 \Uparrow \sigma_2$, then $(\sigma_1 \cup \sigma_2, [\tau_1 \mid \tau_2]) \in [\![P]\!]$ and $(\sigma_1 \cup \sigma_2, [\tau_2 \mid \tau_1]) \in [\![P]\!]$.

RACE If $P = P_1 \uplus P_2$, $\neg(P_1$ **co** $P_2)$, $res(P_1) \cap res(P_2) = \{\}$, $(\sigma_1, \tau_1) \in [\![P_1]\!]$, $(\sigma_2, \tau_2) \in [\![P_2]\!]$, and $\sigma_1 \Uparrow \sigma_2$, then $(\sigma_1 \cup \sigma_2, \top) \in [\![P]\!]$.

Explanation

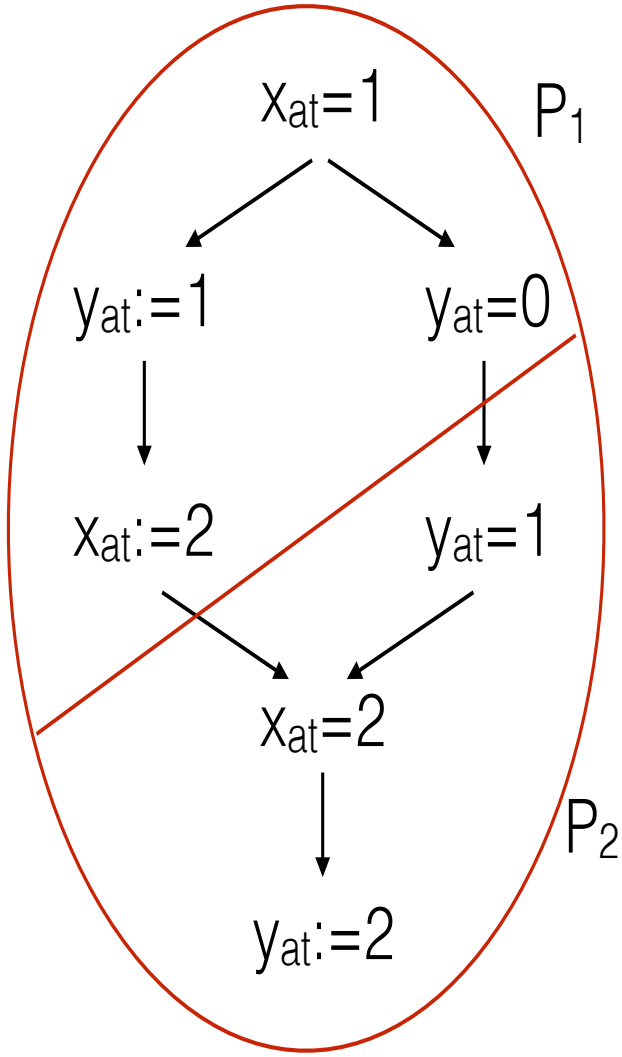
SEQ embodies ***sequential composition*** of footsteps

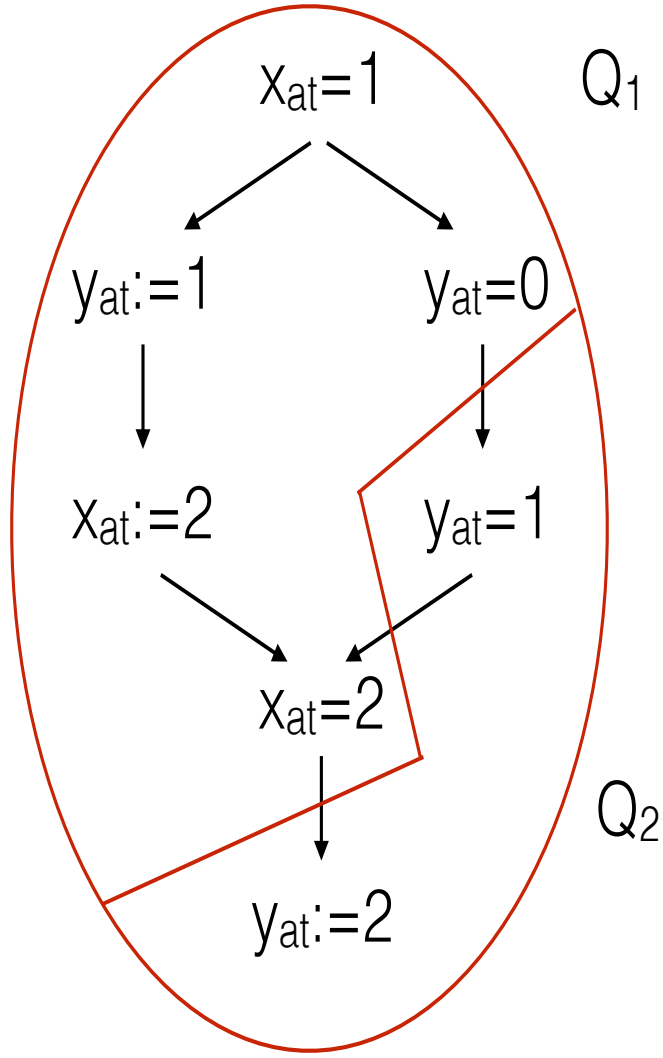
- (σ_1, τ_1) can be followed by (σ_2, τ_2) iff $[\sigma_1 \mid \tau_1] \Uparrow \sigma_2$
- Their cumulative footprint is represented by
  $(\sigma_1 \cup (\sigma_2 \backslash \mathit{dom}\, \tau_1), [\tau_1 \mid \tau_2])$

PAR and RACE enforce a form of
race-detecting ***concurrent composition*** of footsteps

- (σ_1, τ_1) can be composed with (σ_2, τ_2) iff $\sigma_1 \Uparrow \sigma_2$
- Their cumulative footprint is represented by
  $\{(\sigma_1 \cup \sigma_2, [\tau_1 \mid \tau_2]), (\sigma_1 \cup \sigma_2, [\tau_2 \mid \tau_1])\}$ when non-racy,
  $(\sigma_1 \cup \sigma_2, \top)$ when racy
- Requirement that $\mathit{res}(P_1) \cap \mathit{res}(P_2) = \{\}$ only allows concurrent footsteps using distinct locks.
- Concurrent atomic writes to the same variable are allowed, but use of $[\tau_1 \mid \tau_2]$ or $[\tau_2 \mid \tau_1]$ linearizes their effect

Initial Segments

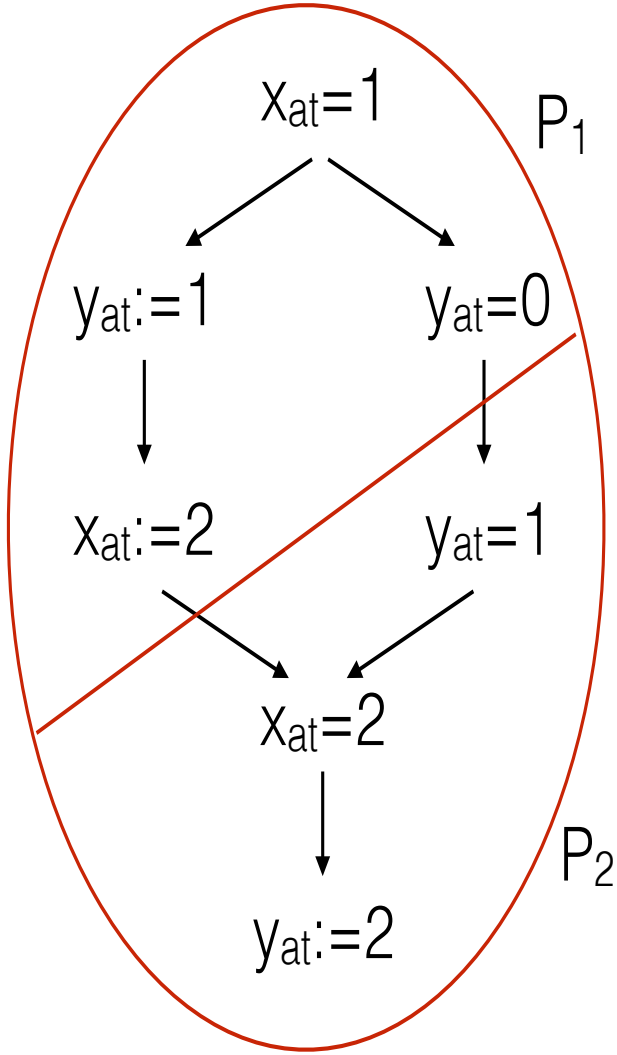




$P = P_1 \triangleright P_2$

$P \neq Q_1 \triangleright Q_2$

Sequencing



footprints

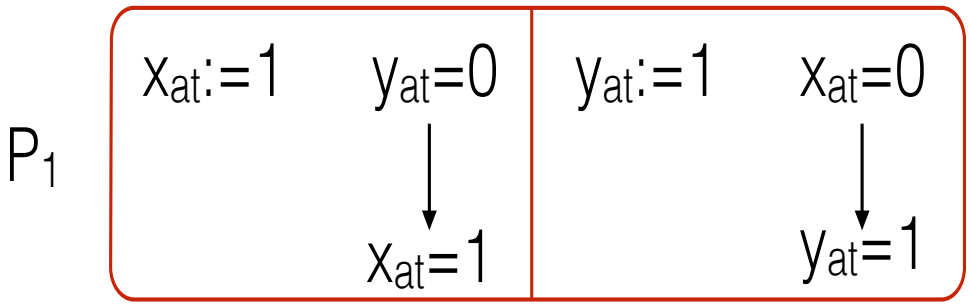
$$\frac{([x{:}0,y{:}0],\,[x{:}2,y{:}1]) \in [\![P_1]\!] \qquad ([x{:}2,y{:}1],\,[y{:}2]) \in [\![P_2]\!]}{([x{:}0,y{:}0],\,[x{:}2,y{:}2]) \in [\![P]\!]}\ \textbf{SEQ}$$

$$P = P_1 \triangleright P_2$$

execution

$$[x{:}0,\ y{:}0] \xRightarrow{P} [x{:}2,\ y{:}2]$$

Concurrence



$P = P_1 \uplus P_2$

$$\frac{P_1 \perp P_2 \quad ([x:0,y:0], [x:1]) \in [\![P_1]\!] \quad ([x:0,y:0], [y:1]) \in [\![P_2]\!]}{([x:0,y:0], [x:1,y:1]) \in [\![P_1 \uplus P_2]\!]}\ \textbf{PAR}$$

footprints

$$[x:0, y:0] \xRightarrow{P} [x:1, y:1]$$

execution

Composing Footprints

Sequential

$$\frac{(\sigma_1, \tau_1) \in [\![P_1]\!] \quad P = P_1 \triangleright P_2 \quad (\sigma_2, \tau_2) \in [\![P_2]\!] \quad [\sigma_1|\tau_1] \Uparrow \sigma_2}{(\sigma_1 \cup \sigma_2 \backslash \tau_1, [\tau_1|\tau_2]) \in [\![P]\!]} \textbf{ SEQ}$$

Concurrent

$$\frac{(\sigma_1, \tau_1) \in [\![P_1]\!] \quad P_1 \perp P_2 \quad (\sigma_2, \tau_2) \in [\![P_2]\!] \quad \sigma_1 \Uparrow \sigma_2}{\begin{array}{l}(\sigma_1 \cup \sigma_2, [\tau_1|\tau_2]) \in [\![P_1 \uplus P_2]\!] \\ (\sigma_1 \cup \sigma_2, [\tau_2|\tau_1]) \in [\![P_1 \uplus P_2]\!]\end{array}} \textbf{ PAR}$$

Examples

- The pomset

$$\begin{array}{cc} x_{at}{:=}1 & y_{at}{:=}1 \\ \downarrow & \downarrow \\ y_{at}{=}0 & x_{at}{=}0 \end{array}$$

  has footprint

$$\{([x:0,y:0],[x:1,y:1])\}$$

- The pomset

$$\begin{array}{cc} x_{at}{:=}1 & x_{at}{:=}2 \\ \downarrow & \\ x_{at}{=}2 & \end{array}$$

  has footprint

$$\{([x:v],[x:2]) \mid v \in V_{int}\}$$

Exercise

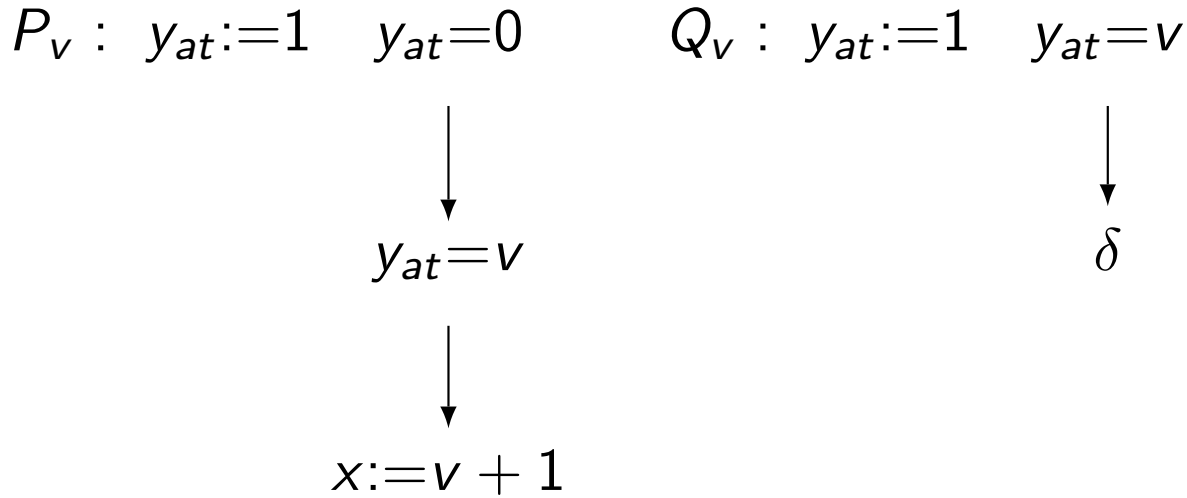
The program

$$y_{at}{:=}1 \parallel \textbf{if } y_{at}{=}0 \textbf{ then } x{:=}y_{at}+1 \textbf{ else skip}$$

has pomsets

$$\{P_v \mid v \in V_{int}\} \cup \{Q_v \mid v \neq 0\},$$

where



Calculate the footprints $[\![P_v]\!], [\![Q_v]\!]$ of these pomsets.

Pomset Execution

Definition

For a finite pomset $(P, <)$ we define the set

$$\mathcal{E}(P) \subseteq \Sigma \times \Sigma^{\top}$$

of *executions* of P, to be:

$$\mathcal{E}(P) = \{(\sigma, [\sigma \mid \tau_1]) \mid \exists \sigma_1 \subseteq \sigma.\ (\sigma_1, \tau_1) \in [\![P]\!]\},$$

where we let $[\sigma \mid \top] = \top$.

- When $(\sigma, \sigma') \in \mathcal{E}(P)$ there is a (race-free) execution of P from σ that respects $<$ and ends in σ'.

  In a race-free execution, each action occurrence of P happens at some finite stage.
- When $(\sigma, \top) \in \mathcal{E}(P)$ there is an execution of (an initial segment of) P from σ that leads to a race condition.

Pomset Execution

Properties

Justification

- Each execution $(\sigma, \sigma') \in \mathcal{E}(P)$ is "justified" by a footprint $(\sigma_1, \tau_1) \in [\![P]\!]$, such that $\sigma_1 \subseteq \sigma \;\&\; \sigma' = [\sigma \mid \tau_1]$.
- Footprints are derived using $\textsc{Act}$, $\textsc{Seq}$, $\textsc{Par}$, $\textsc{Race}$.

Sequencing

- If $P = P_1 \triangleleft P_2$, $(\sigma, \sigma') \in \mathcal{E}(P_1)$ and $(\sigma', \sigma'') \in \mathcal{E}(P_2)$, then $(\sigma, \sigma'') \in \mathcal{E}(P)$.
- If $P = P_1 \triangleleft P_2$ and $(\sigma, \top) \in \mathcal{E}(P_1)$, then $(\sigma, \top) \in \mathcal{E}(P)$.

Concurrence

- If $P = P_1 \| P_2$, $(\sigma, \sigma_1) \in \mathcal{E}(P_1)$ and $(\sigma, \sigma_2) \in \mathcal{E}(P_2)$, it does not follow that $(\sigma, [\sigma_1 \mid \sigma_2]) \in \mathcal{E}(P)$.

Execution Properties

- Footprint derivations, and executions, have a natural *direction*
  - from initial state, toward final state
  - control flow is irreversible
- Initial read actions must "read from" initial state.
- An action can only "happen" after the actions that precede it.
- The structure of a footprint derivation may imply a "happens-before" constraint, e.g.

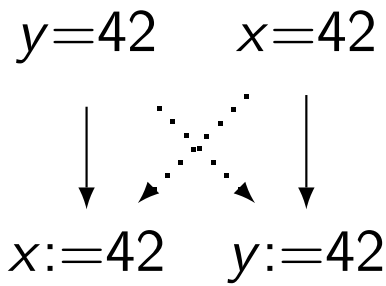
  In any execution of P from σ,
  λ_1 must get executed before λ_2.

- All state changes come from actions of P:

  If $(\sigma, \sigma') \in \mathcal{E}(P)$ and $\sigma'(i) = v' \neq \sigma(i)$,
  there is an occurrence of $i_\alpha{:=}v$ in P.
- Hence, **no *out-of-thin-air writes***.

Out-of-thin-air

In some weak memory model specifications the execution graph



(dots indicate reads-from edges, modification order is trivial)
has an execution from $[x : 0, y : 0]$ to $[x : 42, y : 42]$.

As if each thread "speculates" about its read, then "validates" the guess of the other thread. The writes using 42 come from thin air!

For the underlying pomset (without dots) we have

$$\mathcal{E}(P) = \{(\sigma, \sigma) \mid [x : 42, y : 42] \subseteq \sigma\}$$

It's good that our execution notion excludes out-of-thin-air.

Execution Structure

Theorem

Every (finite) execution $(\sigma, \sigma') \in \mathcal{E}(P)$ *is expressible as a sequence of phases*

$$\sigma = \sigma_0 \xRightarrow{P_0} \sigma_1 \xRightarrow{P_1} \sigma_2 \ \cdots \ \sigma_n \xRightarrow{P_n} \sigma_{n+1} = \sigma'$$

where $P = P_0 \triangleleft P_1 \ldots \triangleleft P_n$ *and each phase is justified by footprint, i.e. for each* j $\exists(\tau_j, \tau_j') \in [\![P_j]\!]$ *with* $\tau_j \subseteq \sigma_j$ *and* $\sigma_{j+1} = [\sigma_j \mid \tau_j']$.

- Each phase performs an *initial segment* of the rest of P, until no actions remain or a race is detected.
- Within a phase, writes to distinct variables may happen independently, so no total store order on *all* writes.
- Footprint rules ACT, SEQ, PAR, RACE allow true concurrency but still guarantee a total store order *per* single variable.

Infinite Executions

Execution extends to an infinite pomset P as follows:

Definition

$(\sigma, \bot) \in \mathcal{E}(P)$ iff there is a sequence of finite pomsets P_n such that

$$P = P_0 \lhd P_1 \ldots \lhd P_n \cdots$$

and a sequence of states σ_n $(n \geq 0)$ such that

$$\sigma = \sigma_0 \stackrel{P_0}{\Longrightarrow} \sigma_1 \stackrel{P_1}{\Longrightarrow} \sigma_2 \ \cdots \ \sigma_n \stackrel{P_n}{\Longrightarrow} \sigma_{n+1} \ \cdots$$

This deals naturally with infinite executions,
and builds in (weak, process) **fairness** *automatically.*

Program Behavior

Definition

We define *program footprints*

$$[\![c]\!] = \bigcup \{ [\![P]\!] \mid P \in \mathcal{P}(c) \}$$

and *program executions*

$$\mathcal{E}(c) = \bigcup \{ \mathcal{E}(P) \mid P \in \mathcal{P}(c) \}$$

in the obvious way.

Example

$$[\![\textbf{resource } r \textbf{ in } (inc \| inc)]\!] = \{([x : v], [x : v + 2]) \mid v \in V_{int}\}$$

$$\mathcal{E}(\textbf{resource } r \textbf{ in } (inc \| inc)) = \{(\sigma, [\sigma \mid x : v + 2]) \mid x : v \in \sigma\}$$

Program Analysis, revisited

To determine which weak memory executions are possible for a given program:

Generate the pomsets of the program

Determine which ones are executable from σ.

Extract the final state σ'.

Still requires combinatorial analysis:

- how to decompose P into executable chunks
- focus on "reachable" cross-sections of P

But many different decompositions will yield the same execution pairs, and we can exploit pomset structure to simplify analysis.

Results

On these litmus tests, *pomset execution* $\mathcal{E}$
yields behaviors consistent with rel/acq

Litmus Test	SC	TSO	rel/acq	$\mathcal{E}$
1. **Store Buffering**	✗	✓	✓	✓
2. **Message Passing**	✓	✓	✓	✓
3. **IRIW**	✗	✗	✓	✓
4. **Coherence**	✓	✓	✓	✓
5. **Optimization**	✓	✓	✓	✓

Litmus Test 1

Store Buffering

SC ✗ **TSO ✓** **rel/acq ✓**

$$(x_{at}{:=}1; z_1{:=}y_{at}) \parallel (y_{at}{:=}1; z_2{:=}x_{at})$$

- The pomset

$$\{x_{at}{:=}1\ y_{at}{=}0\ z_1{:=}0, y_{at}{:=}1\ x_{at}{=}0\ z_2{:=}0\}$$

  is executable from $[x : 0, y : 0, z_1 : v_1, z_2 : v_2]$
  and terminates in $[x : 1, y : 1, z_1 : 0, z_2 : 0]$.
- Justification: use PAR and the footprint entries

$$([x : 0, y : 0, z_1 : v_1], [x : 1, z_1 : 0]) \in [\![\{x_{at}{:=}1\ y_{at}{=}0\ z_1{:=}0\}]\!]$$
$$([x : 0, y : 0, z_2 : v_2], [y : 1, z_2 : 0]) \in [\![\{y_{at}{:=}1\ x_{at}{=}0\ z_2{:=}0\}]\!]$$

- In this execution the reads see stale values. ✓

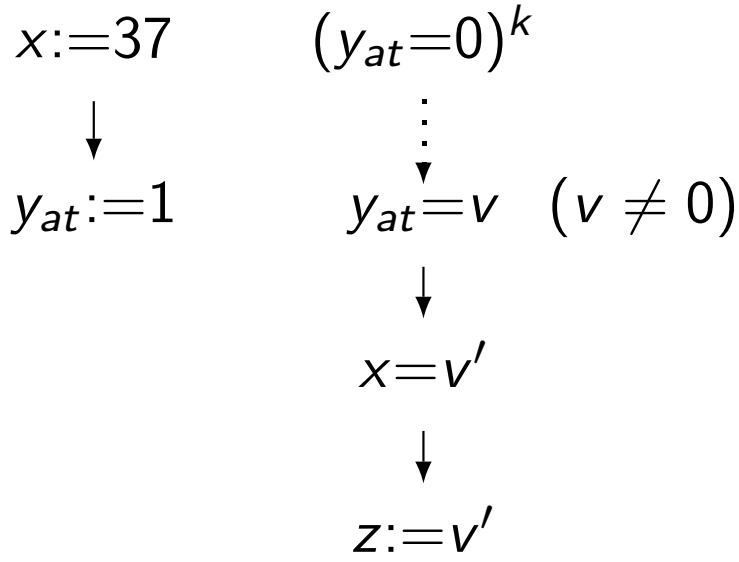
Litmus Test 2

Message Passing

SC ✓ TSO ✓ rel/acq ✓

$$(x{:=}37; y_{at}{:=}1) \parallel (\textbf{while } y_{at} = 0 \textbf{ do skip};\ z{:=}x)$$

- Each finite pomset has the form



  Only executable from $[x : 0, y : 0, z : 0]$ for $v = 1, v' = 37$ ✓

- The infinite pomset $\{x{:=}37\ y_{at}{:=}1,\ (y_{at}{=}0)^{\omega}\}$

  is *not* executable from $[x : 0, y : 0]$. Executions are fair! ✓

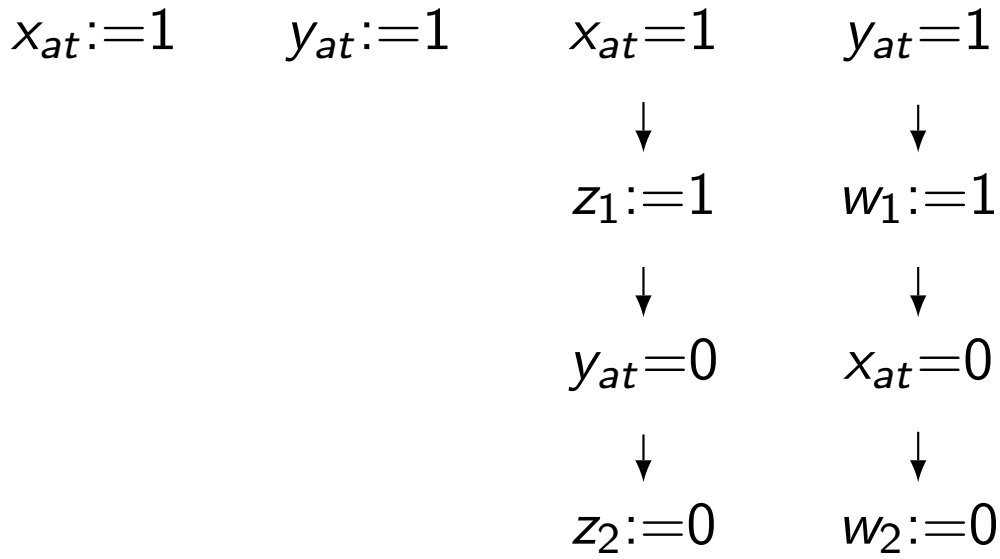
Litmus Test 3

Independent Reads of Independent Writes

SC ✗ TSO ✗ rel/acq ✓

$$x_{at}{:=}1 \parallel y_{at}{:=}1 \parallel (z_1{:=}x_{at};\ z_2{:=}y_{at}) \parallel (w_1{:=}y_{at};\ w_2{:=}x_{at})$$

- The pomset



  is executable from $[x : 0, y : 0, \ldots]$ ✓

- Threads see the writes to x and y in different orders.

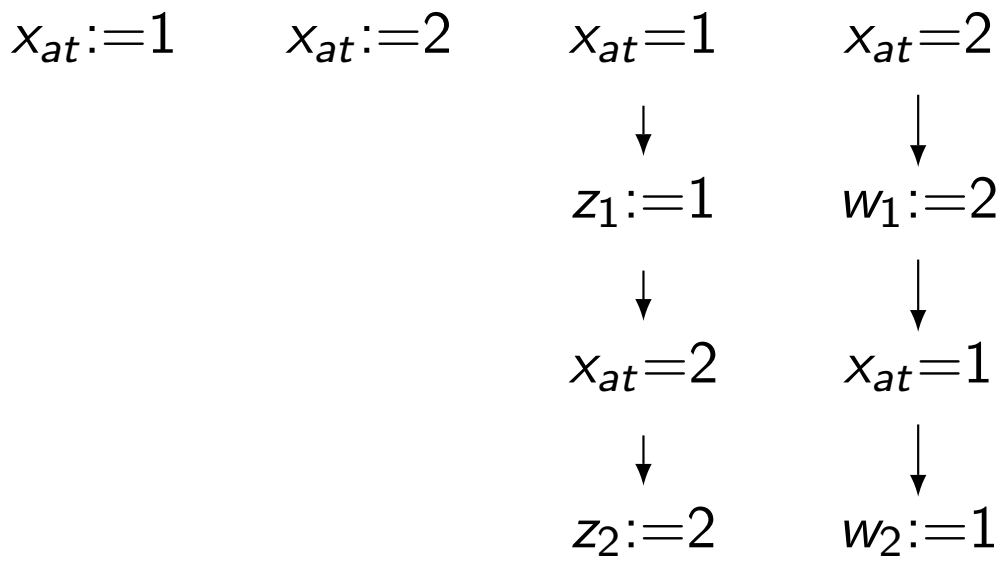
Litmus Test 4

Coherence

SC ✓ TSO ✓ rel/acq ✓

$$x_{at}{:=}1 \parallel x_{at}{:=}2 \parallel (z_1{:=}x_{at};\ z_2{:=}x_{at}) \parallel (w_1{:=}x_{at};\ w_2{:=}x_{at})$$

- The pomset



  is not executable from $[x : 0, y : 0, \ldots]$.

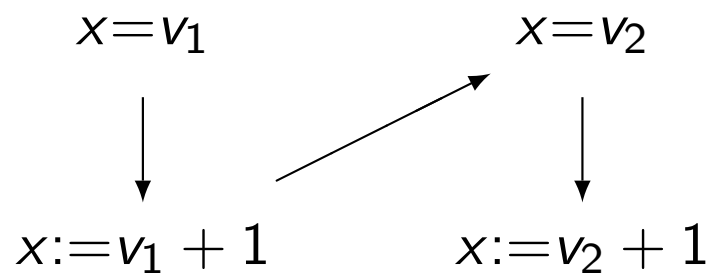
- Writes to x appear in the same order, to all threads. ✓

Concurrent increments using a private lock

- Let *inc* be **lock** r; $x{:=}x+1$; **unlock** r.
- The pomsets of

$$\textbf{resource } r \textbf{ in } (inc \parallel inc)$$

have form



where $v_1, v_2 \in V_{int}$

- Only *executable from* $[x : v]$ when $v_1 = v, v_2 = v + 1$, so

$$\mathcal{E}(\textbf{resource } r \textbf{ in } (inc \parallel inc)) = \mathcal{E}(x{:=}x+2)$$ ✓

Litmus Test 5

Optimization

SC ✓ TSO ✓ rel/acq ✓

Non-atomic code can be re-ordered, in a race-free program context, without affecting execution.

Theorem

If c_1 and c_2 are non-atomic and $[\![c_1]\!] = [\![c_2]\!]$, then $\mathcal{E}(C[c_1]) = \mathcal{E}(C[c_2])$.

Examples

$$\begin{array}{rl} & \mathcal{E}(c \parallel \textbf{lock}\ r; x{:=}x+1; x{:=}x+1; \textbf{unlock}\ r) \\ = & \mathcal{E}(c \parallel \textbf{lock}\ r; x{:=}x+2; \textbf{unlock}\ r) \end{array}$$

$$\begin{array}{rl} & \mathcal{E}(c \parallel c_{11}; x{:=}1; y{:=}2; c_{12}) \\ = & \mathcal{E}(c \parallel c_{21}; y{:=}2; x{:=}1; c_{22}) \end{array}$$

Semantic Properties

- $\mathcal{P}$ is *compositional*
  - supports sytax-directed reasoning
- $\mathcal{E}$ is *(co-)inductively defined*, based on $\mathcal{P}$
  - supports computational reasoning
- Both $\mathcal{P}$ and $\mathcal{E}$ are *succinct*
  - avoids combinatorial explosion
- Definition of $\mathcal{E}$ builds in *fairness*
  - supports liveness analysis

Example

Let $inc^{(n)}$ be $inc \| \cdots \| inc$ (n times).

$$\begin{array}{rcl} \mathcal{P}(\textbf{resource } r \textbf{ in } inc^{(n)}) & = & \mathcal{P}(x{:=}x+1;\ \cdots\ ; x{:=}x+1) \\ \mathcal{E}(\textbf{resource } r \textbf{ in } inc^{(n)}) & = & \{(\sigma, [\sigma \mid x : v+n]) \mid x : v \in \sigma\} \end{array}$$

Pomset Equivalence

- Pomsets are *equivalent* if they are *order-isomorphic*, allowing for elision of δ actions
- Lift to sets of pomsets in the obvious way.
- We have standard laws, such as

$$\begin{array}{rcl} c_1 \| (c_2 \| c_3) & \equiv_{\mathcal{P}} & (c_1 \| c_2) \| c_3 \\ c_1 \| c_2 & \equiv_{\mathcal{P}} & c_2 \| c_1 \\ c_1 ; (c_2 ; c_3) & \equiv_{\mathcal{P}} & (c_1 ; c_2) ; c_3 \\ c ; \mathbf{skip} & \equiv_{\mathcal{P}} & \mathbf{skip} ; c \equiv_{\mathcal{P}} c \\ c \| \mathbf{skip} & \equiv_{\mathcal{P}} & c \end{array}$$

and *scope contraction*

$$\mathbf{resource}\ r\ \mathbf{in}\ (c_1 \| c_2) \equiv_{\mathcal{P}} (\mathbf{resource}\ r\ \mathbf{in}\ c_1) \| c_2 \quad \text{when } r \text{ not free in } c_2$$

- Proofs of validity are *easier* than for trace semantics!

Execution Equivalence

- We define *execution equivalence* for programs by

$$c_1 \equiv_{\mathcal{E}} c_2 \text{ iff } \mathcal{E}(c_1) = \mathcal{E}(c_2)$$

- Pomset equivalence implies execution equivalence

$$c_1 \equiv_{\mathcal{P}} c_2 \text{ implies } c_1 \equiv_{\mathcal{E}} c_2$$

- So we also have

$$c_1 \| (c_2 \| c_3) \quad \equiv_{\mathcal{E}} \quad (c_1 \| c_2) \| c_3$$

and *scope contraction*

$$\textbf{resource } r \textbf{ in } (c_1 \| c_2) \equiv_{\mathcal{E}} (\textbf{resource } r \textbf{ in } c_1) \| c_2$$
$$\text{when } r \text{ not free in } c_2$$

Pomsets and Execution Graphs

Pomset executions, based on our denotational framework, may be used to extract **execution graphs**:

An alternative to the operational/axiomatic approach

Theorem

- *Given (a derivation for) an execution* $(\sigma, \sigma') \in \mathcal{E}(P)$*, we can extract happens-before, reads-from, modification-order relations on the action occurrences in* P*.*
- *This produces an execution graph consistent with* (σ, σ')*.*
- *Properties* (1)–(4) *hold, suitably formalized.*

Sketch

Use the phase structure of a pomset execution,
and the inductive characterization of footprints.

Advantages

In contrast with the operational/axiomatic approach

- No need for complex axioms.
- Not necessary to assume knowledge of entire program.
- No need to deal explicitly with multiple relations
  happens-before, *reads-from*, *modification-order*
  We just use program order $<$ and can *derive* relations with the required properties, from the phase structure of an execution.
- We also handle programs with infinite behaviors.

Limitations

- We dealt with a weak memory model that we characterized only implicitly, and only in *abstract* terms

  - actually, we see this as an advantage!

- Our WMM is closely related to a fragment of C11

  - similar to release/acquire, but there are subtle differences
  - C11 not really stable...
  - our WMM seems to coincide with Vafeiadis' recently proposed revision to C11 release/acquire axioms

- Would be interesting to establish formal connection.

- We only distinguished between *at* and *na*.
- To extend, would need wider range of atomicity levels, e.g.

  sc (Java volatile)

  This would be straightforward, semantically.
  But requires development of more complex execution models.

Pomsets and True Concurrency

"We are not the first to advocate partial-order semantics."
Pratt 1986

- Pratt's *pomsets* form a "true concurrency" *process algebra*

  But *too* abstract, with no notions of state, effect, execution
  No *locally finite height* requirement (leads to "semantic junk")
  Not *fair*, despite being "*sine qua non* among theoreticians"

- Mostly concerned with abstract properties, e.g.
  Every poset is representable as the set of its linearizations.

  But this fails when we look at *execution*, because of PAR

$$\mathcal{E}(P) \neq \bigcup \{\mathcal{E}(P') \mid P' \in \textsc{Lin}(P)\}$$

- "Operational semantics. . . forces an interleaving view".

  But we *can* give a "true concurrency" operational semantics.

Prior Related Work

"We are not the first to advocate partial-order semantics."
Pratt 1986

- We are not even the *second*!
- Pioneering work by Petri, Mazurkiewicz, . . . , Winskel
  Petri nets, Mazurkiewicz traces,. . . , event structures
  on partial-order semantics for SC notions of concurrency


Petri


Mazurkiewicz


Winskel

Current Related Work

Recent work shows renewed interest in and relevance of partial-order models, in weak memory settings:

- *Brookes Is Relaxed, Almost!*
  R. Jagadeesan, G. Petri, J. Riely.
  - adapts "transition traces" from SC to TSO
- *Relaxed Memory Models: an Operational Approach*
  G. Boudol, G. Petri.
  - interleaving, but distributed state with per-thread buffers
- *Weak memory models using event structures*
  S. Castellan.

Conclusions

"The one duty we owe to history is to rewrite it." ***Oscar Wilde***

A denotational *true concurrency* framework for *weak memory*

- pomset semantics + execution

Supports *compositional* reasoning

- race-free partial correctness, safety and liveness
- fairness comes for free

Should be applicable to other weak memory models

- *same* pomset semantics, *different* execution, ...

May offer a new foundation for weak memory logics and tools

- GPS, Relaxed Separation Logic
- `cppmem`, `diy`, `litmus`, ... Alglave, Sewell, et al.

Future

- Tools for pomset execution
  - partitioning a pomset
  - reasoning about environment
- Explore alternative forms of pomset execution
  - tailored to other weak memory models
- Truly concurrent "transition traces", e.g.

$$(\sigma_0, \sigma_0')(\sigma_1, \sigma_1') \ldots (\sigma_n, \sigma_n')$$

where

$$P = P_0 \lhd P_1 \lhd \cdots \lhd P_n$$

and

$$\sigma_0 \stackrel{P_0}{\Longrightarrow} \sigma_0' \;\&\; \sigma_1 \stackrel{P_1}{\Longrightarrow} \sigma_1' \;\cdots\; \&\; \sigma_n \stackrel{P_n}{\Longrightarrow} \sigma_n'$$

- Infinite traces
  - total correctness, safety and liveness

Summary

- Complexity of reasoning about concurrent programs.

  "The major problem facing software developers. . . "
- Behavior depends on whether threads cooperate or interfere.
- Need compositional reasoning, to exploit modular structure, minimize book-keeping, reduce number of interactions.
- There is a clear gap between theory and practice.

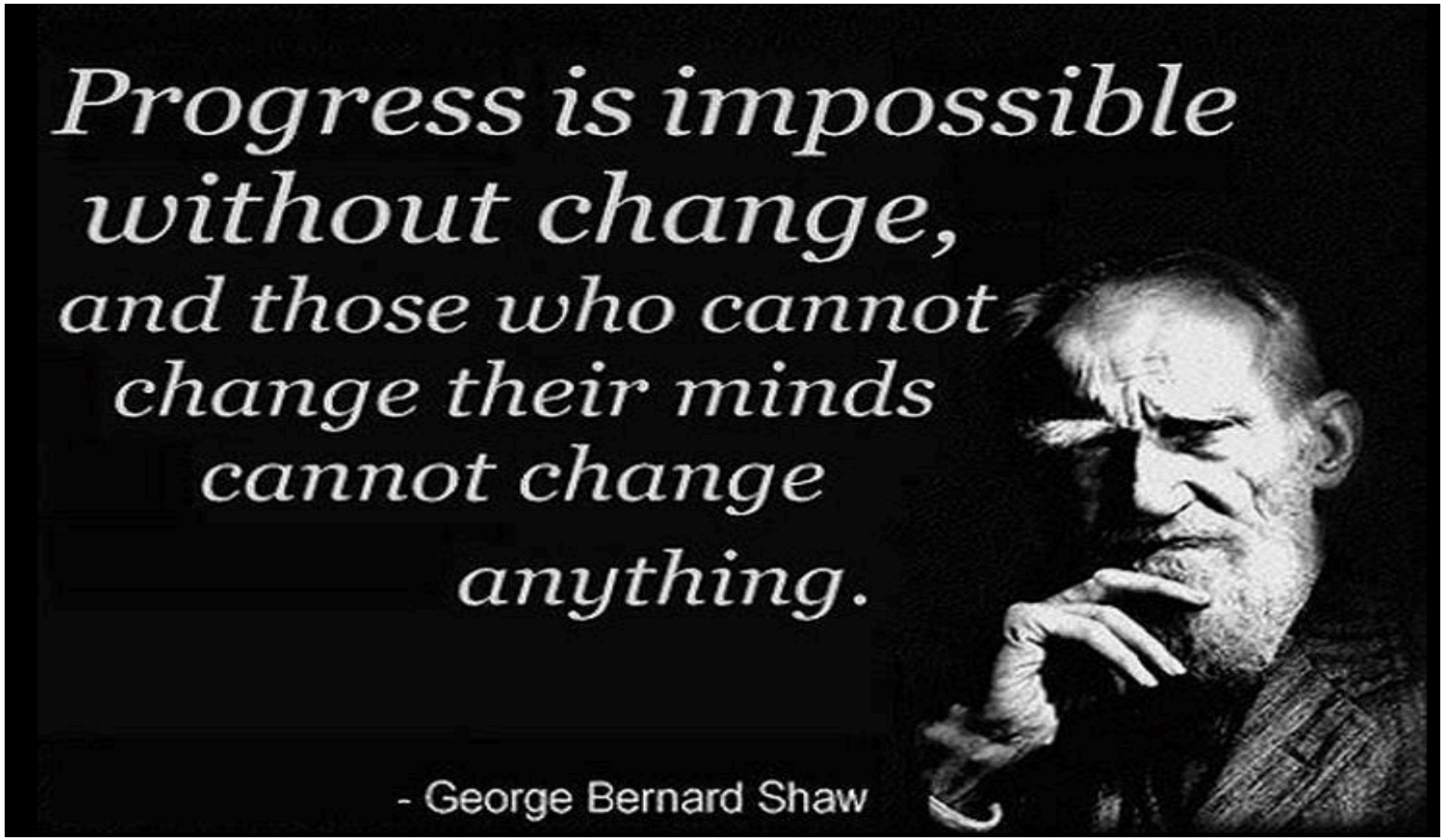
  Formal methods, based on a semantics that assumes sequential consistency, do not account for weak memory models.
- State-of-the-art tools lack generality and scalability.

  Based on operational semantics, inhibits compositionality.
- We offer a denotational framework that can promote compositional reasoning to weak memory models.
- This kind of foundational research is essential if verification technology is to be relevant to real-world programs running on modern architectures.

Citations

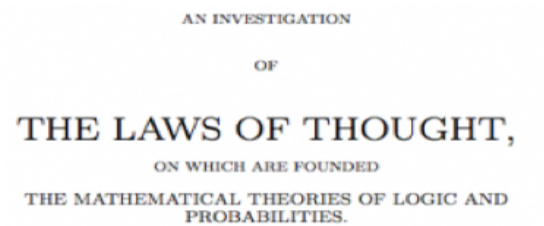
- J. Alglave, L. Maranget, S. Sarkar and P. Sewell. *Litmus: Running Tests Against Hardware*. TACAS, 2011.
- S. Brookes. *A semantics for concurrent separation logic*. Theoretical Computer Science, 2007.
- L. Lamport. *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*. IEEE Trans. Comput. 1979.
- P. O'Hearn. *Resources, concurrency, and local reasoning*. Theoretical Computer Science, 2007.
- D. Park. *On the semantics of fair parallelism*. **Abstract Software Specifications**, Springer LNCS vol. 86, 1979.
- C. Petri. *Kommunikation mit Automaten*. Thesis, U. Bonn, 1962.
- V. Pratt. *Modelling concurrency with partial orders*. IJPP, 1986.
- A. Turon, V. Vafeiadis, D. Dreyer. *GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation*. OOPSLA, 2014.
- V. Vafeiadis and C. Narayan. *Relaxed separation logic: A program logic for C11 concurrency*. OOPSLA, 2013.

Epilogue


Progress is impossible
without change,
and those who cannot
change their minds
cannot change
anything.
- George Bernard Shaw

Inspiration

No matter how correct a mathematical theorem may appear to be, one ought never to be satisfied that there was not something imperfect about it until it also gives the impression of being beautiful.


AN INVESTIGATION

OF

THE LAWS OF THOUGHT,

ON WHICH ARE FOUNDED

THE MATHEMATICAL THEORIES OF LOGIC AND PROBABILITIES.