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...
“Of the many forms of false culture, a premature converse with abstractions is perhaps the most likely to prove fatal . . .”

George Boole (1859)
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

\[
(lock r; x := x + 1; unlock r) \parallel (lock r; y := x + x; unlock r)
\]
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

thread_1

private registers

thread_2

private registers

thread_3

private registers

shared RAM
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

\[ \mathcal{T}(c_1 \parallel c_2) = \bigcup \{ \alpha_1 \parallel \alpha_2 \mid \alpha_1 \in \mathcal{T}(c_1) \land \alpha_2 \in \mathcal{T}(c_2) \} \]

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

\[
\begin{align*}
\langle c_1, \sigma \rangle &\to \langle c'_1, \sigma' \rangle \\
\langle c_1 \parallel c_2, \sigma \rangle &\to \langle c'_1 \parallel c_2, \sigma' \rangle \\
\langle c_2, \sigma \rangle &\to \langle c'_2, \sigma' \rangle \\
\langle c_1 \parallel c_2, \sigma \rangle &\to \langle c_1 \parallel c'_2, \sigma' \rangle
\end{align*}
\]
Program properties such as partial and total correctness, safety and liveness are based on (interference-free) execution, e.g.

\[ \{p\}c\{q\} : \forall \alpha \in \mathcal{T}(c). \text{ if } \sigma \models p \& \sigma \xrightarrow{\alpha} \sigma' \text{ then } \sigma' \models q \]

Fairness crucial for liveness; infinite traces for safety, liveness.

These properties involve \textit{sequentially executable} traces.

To determine the (sequential) traces of \(c_1 \parallel c_2\), need to include non-sequential traces of \(c_1\) and \(c_2\).

The sequential traces of \(c_1 \parallel c_2\) 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

\[ \mathcal{T}(c_1 \parallel c_2) = \bigcup \{\alpha_1 \parallel \alpha_2 \mid \alpha_1 \in \mathcal{T}(c_1) \& \alpha_2 \in \mathcal{T}(c_2)\} \]

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

\[
\begin{align*}
\{p_1\} & c_1 \{q_1\} \quad \{p_2\} & c_2 \{q_2\} \\
\{p_1 \land p_2\} & c_1 \parallel c_2 \{q_1 \land q_2\}
\end{align*}
\]

Owicki, Gries 1976

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

\[
\begin{align*}
\{p_1\} & c_1 \{q_1\} \quad \{p_2\} & c_2 \{q_2\} \\
\{p_1 \star p_2\} & c_1 \parallel c_2 \{q_1 \star q_2\}
\end{align*}
\]

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 \((\sigma, \sigma')\) 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 \((\sigma, \sigma')\) 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**
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?**
Status Quo

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

Alglave, Batty, Sewell, et al

\[(\text{CohRR}) : \neg \exists a, b. \ hb(a, b) \land \ mo(rf(b), rf(a))\]
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** \((\sigma, \sigma')\) is an execution graph whose initial reads are consistent with \(\sigma\), and \(mo\)-final writes determine \(\sigma'\).
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 $(\sigma, \sigma')$.

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

\[(z_1 := x_{acq}; z_2 := y_{acq}) \parallel (y_{rel} := 2; x_{rel} := 1) \parallel (x_{rel} := 2; y_{rel} := 1)\]

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.
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
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

\[ i \in \text{Ide}, \ r \in \text{Res}, \ e \in \text{Exp}_{\text{int}}, \ b \in \text{Exp}_{\text{bool}}, \ c \in \text{Com} \]

\[
\begin{align*}
e & ::= n \mid i_\alpha \mid e_1 + e_2 \\
b & ::= \text{true} \mid \neg b \mid b_1 \lor b_2 \mid e_1 = e_2 \mid e_1 < e_2 \\
c & ::= \text{skip} \mid i_\alpha := e \mid c_1; c_2 \mid c_1 || c_2 \\
& \quad \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \\
& \quad \mid \text{lock } r \mid \text{unlock } r \mid \text{resource } r \text{ in } c \\
\alpha & ::= \text{at} \mid \text{na}
\end{align*}
\]

- Read and write variable occurrences are tagged \( \alpha \)
  - Atomic \( \text{at} \), non-atomic \( \text{na} \)
  - May omit \( \alpha \) when \( \text{na} \)
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
  \( (\text{lock } r ; x_{na} := 1 ; \ldots ) \parallel (\text{lock } r ; y_{na} := x_{na} ; \ldots ) \)

► Blocks are statically scoped
  \( (\text{resource } r \text{ in } c_1 ) \parallel c_2 \)
Outline

- **Litmus tests**
  - 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

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

- 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

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

- 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

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

- 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

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

▶ 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.
If $c$ uses $r$ to access $x$,

$$c \parallel \text{lock } r; x:=x + 1; x:=x + 1; \text{unlock } r$$

$$\equiv c \parallel \text{lock } r; x:=x + 2; \text{unlock } r$$

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

$$c \parallel (c_{11}; x:=1; y:=2; c_{12})$$

$$\equiv c \parallel (c_{21}; y:=2; x:=1; c_{22})$$

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

framework = semantics + 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

\[ \sigma, \tau \in \Sigma = \text{Ide} \rightarrow_{\text{fin}} V_{\text{int}} \]

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

\[
[\chi_1 : v_1, \ldots, \chi_n : v_n] = \{(\chi_i, v_i) \mid 1 \leq i \leq n\}
\]

- We write \([\sigma \mid \tau]\) for the state obtained by updating \(\sigma\) with \(\tau\)

\[
[\sigma \mid \tau] = (\sigma \setminus \text{dom}(\tau)) \cup \tau
\]

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

- States \(\sigma_1\) and \(\sigma_2\) are consistent, written as \(\sigma_1 \uparrow \sigma_2\), iff they agree on \(\text{dom}(\sigma_1) \cap \text{dom}(\sigma_2)\), i.e.

\[
\forall i \in \text{dom}(\sigma_1) \cap \text{dom}(\sigma_2). \quad \sigma_1(i) = \sigma_2(i).
\]

When this happens, we have

\[
[\sigma_1 \mid \sigma_2] = [\sigma_2 \mid \sigma_1] = \sigma_1 \cup \sigma_2 \in \Sigma.
\]
\(\lambda, \mu \in \Lambda\)

\[
\begin{align*}
\lambda & ::= \delta | i_{\alpha} = v | i_{\alpha} := v | \text{lock } r | \text{unlock } r \\
\alpha & ::= \text{at} | \text{na}
\end{align*}
\]

- \(\delta\) is an idle action.
- Reads \(i_{\alpha} = v\) and writes \(i_{\alpha} := v\) are tagged as atomic \(\text{at}\) or non-atomic \(\text{na}\). We may omit \(\alpha\) when non-atomic.
- The \text{lock} and \text{unlock} actions are atomic.
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 \( \beta \) onto \( \alpha \)

\[ \alpha \beta = \alpha \quad \text{if } \alpha \text{ infinite} \]

Examples

\[ x_{at} := 1 \quad x_{at} = 1 \quad y := 1 \]
\[ x_{at} := 1 \quad x_{at} = 0 \quad y := 0 \]

*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

\[ [\lambda] \subseteq \Sigma \times \Sigma^T \]

is given by

\[ [\delta] = \{([[ ]], [ ]]) \} \]
\[ [i_\alpha:=v] = \{([[i : v], [ ]]) \} \]
\[ [i_\alpha:=v] = \{([[i : v_0], [i : v]]) \mid v_0 \in V_{int} \} \]
\[ [锁 r] = \{([[r : 0], [r : 1]]) \} \]
\[ [解锁 r] = \{([[r : 1], [r : 0]]) \} \]

- Describes minimal state needed by, and affected by, an action.
- \( \lambda \) is enabled in \( \sigma \) 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 \( \lambda \) reads \( \sigma_1 \), writes \( \tau_1 \).
Effects

Definition
The effect of an action

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

is given by

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

Theorem
Actions have the following effects:

\[ \mathcal{E}(\delta) = \{ (\sigma, \sigma) | \sigma \in \Sigma \} \]
\[ \mathcal{E}(i_\alpha = \nu) = \{ (\sigma, \sigma) | i : \nu \in \sigma \} \]
\[ \mathcal{E}(i_\alpha := \nu) = \{ (\sigma, [\sigma | i : \nu]) | i \in \text{dom}(\sigma) \} \]
\[ \mathcal{E}(\text{lock } r) = \{ (\sigma, [\sigma | r : 1]) | r : 0 \in \sigma \} \]
\[ \mathcal{E}(\text{unlock } r) = \{ (\sigma, [\sigma | r : 0]) | r : 1 \in \sigma \} \]
Sequential execution

For actions and traces, we define:

\[ \sigma \xrightarrow{\lambda} \sigma' \quad \text{iff} \quad (\sigma, \sigma') \in \mathcal{E}(\lambda) \]

\[ \sigma \xrightarrow{\lambda_1 \cdots \lambda_n} \sigma' \quad \text{iff} \quad \sigma \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_n} \sigma' \]

Basic facts:

\[ \sigma \xrightarrow{\delta} \sigma \quad \text{always} \]

\[ \sigma \xrightarrow{i_\alpha = v} \sigma \quad \text{iff} \quad (i, v) \in \sigma \]

\[ \sigma \xrightarrow{i_\alpha ::= v} \sigma' \quad \text{iff} \quad i \in \text{dom}(\sigma) \& \sigma' = [\sigma \mid i : v] \]

\[ \sigma \xrightarrow{\text{lock } r} \sigma' \quad \text{iff} \quad \sigma(r) = 0 \& \sigma' = [\sigma \mid r : 1] \]

\[ \sigma \xrightarrow{\text{unlock } r} \sigma' \quad \text{iff} \quad \sigma(r) = 1 \& \sigma' = [\sigma \mid r : 0] \]

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

**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 \(\text{Pom}\) 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 \(\Phi : V \rightarrow \Lambda\).

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

Example

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

\( \lambda_1 \)
\( \lambda_2 \)
\( \lambda \)
\( \mu \)

Transitivity edges omitted...
Locally Finite Height

Not allowed

\[ \lambda_1 \rightarrow \lambda_2 \rightarrow \cdots \rightarrow \lambda_n \rightarrow \cdots \rightarrow \mu \]

\[ \lambda_1 \rightarrow \lambda_2 \rightarrow \cdots \rightarrow \lambda_n \rightarrow \cdots \rightarrow \mu \]

\[ \mu \text{ not finitely reachable} \]
Locally Finite Height

Allowed

Infinite total height, but each $\lambda_n$ 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 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

\[
\begin{align*}
\text{lock } r \\
\downarrow \\
\lambda_1 \\
\downarrow \\
\text{unlock } r
\end{align*}
\quad
\begin{align*}
\text{lock } r \\
\downarrow \\
\lambda_2 \\
\downarrow \\
\text{unlock } r
\end{align*}
\quad
\begin{align*}
\text{lock } r \\
\downarrow \\
\lambda_1 \\
\downarrow \\
\text{unlock } r \\
\downarrow \\
\lambda_2 \\
\downarrow \\
\text{unlock } r
\end{align*}
\]
Building Pomsets

cf. Pratt 1986

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 $P_2$

$$ (P_1, <_1); (P_2, <_2) = (P_1, <_1) $$

if $|P_1|$ is infinite

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

if $|P_1|$ is finite

These operations extend pointwise to sets of pomsets.

---

$^3$Pratt’s $P_1; P_2$ does not handle the infinite case separately.
Sequential composition

$P$

$a := 1$

$x_a := 1 \quad y_a := 1$

$P; P$

$a := 1$

$x_a := 1 \quad y_a := 1$

$a := 1$

$x_a := 1 \quad y_a := 1$
Parallel composition

\[ P \parallel P \]

\[ a := 1 \]

\[ x_{at} := 1 \quad \quad y_{at} := 1 \]

\[ a := 1 \]

\[ x_{at} := 1 \]

\[ y_{at} := 1 \quad x_{at} := 1 \]

\[ y_{at} := 1 \]
Pomset Properties

\[ P_1; (P_2; P_3) = (P_1; P_2); P_3 \]
\[ P_1 \parallel (P_2 \parallel P_3) = (P_1 \parallel P_2) \parallel P_3 \]
\[ P_1 \parallel P_2 = P_2 \parallel P_1 \]
\[ P \parallel P \neq P \]
\[ (P_1 \parallel P_2); P_3 \neq (P_1; P_3) \parallel (P_2; P_3) \]
\[ P_1; (P_2 \parallel P_3) \neq (P_1; P_2) \parallel (P_1; P_3) \]

- \( P = Q \) 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.
  
  $P^0 = \{\delta\}$
  
  $P^{k+1} = P \cdot P^k$

- Let $P^\omega$ be the countably infinite sequential composition.

- Let $P^* = \bigcup_{n=0}^{\infty} P^n$.

Example

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

$P^\omega : x_{at} := 1 \quad y_{at} := 1$

\[\begin{array}{c}
\downarrow \\
\times \\
\downarrow \\
\times \\
\downarrow \\
\times \\
\vdots \\
\vdots 
\end{array}\]

$P^0 : x_{at} := 1 \quad y_{at} := 1$

$P^1 : x_{at} := 1 \quad y_{at} := 1$

$\vdots$
Pomset Extension

Definition

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

Example

- $P^\omega$ extends $P$

\[
P : \quad x_{at} := 1 \quad y_{at} := 1 \quad P^\omega : \quad x_{at} := 1 \quad y_{at} := 1
\]

\[
\begin{array}{c}
\downarrow \\
\times \\
\downarrow \\
\times \\
\downarrow \\
\vdots
\end{array}
\]

\[
x_{at} := 1 \quad y_{at} := 1
\]

\[
\begin{array}{c}
\downarrow \\
\times \\
\downarrow \\
\times \\
\downarrow \\
\vdots
\end{array}
\]
Exercise

- Find all pomset extensions of

\[
\begin{align*}
\text{lock } r & & \text{lock } r \\
\downarrow & & \downarrow \\
\lambda_1 & & \lambda_2 \\
\downarrow & & \downarrow \\
\text{unlock } r & & \text{unlock } r
\end{align*}
\]

that are linear for \( r \).

- Which of these are sequentially executable from \([r : 0]\)?

(Assume that the actions \(\lambda_1\) and \(\lambda_2\) do not involve \( r \).)
Pomset operations

- Let \( \mathcal{P}(\text{Pom}) \) be the powerset of \( \text{Pom} \). Programs denote sets of pomsets.
- \( \mathcal{P}(\text{Pom}) \) is a complete lattice, ordered by set inclusion.
- Pomset operations extend to sets of pomsets:
  - For \( X, Y \subseteq \text{Pom} \) we write
    \[
    X; Y = \{ P; Q \mid P \in X, Q \in Y \}
    \]
    \[
    X \parallel Y = \{ P \parallel Q \mid P \in X, Q \in Y \}
    \]
- Similarly for iteration:
  \[
  X^n = \{ P_1; \cdots ; P_n \mid P_i \in X \}
  \]
  \[
  X^\omega = \{ P_1; \cdots ; P_n; \cdots \mid P_i \in X \}
  \]
  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

\( \mathcal{P} : \text{Exp}_{\text{int}} \rightarrow \mathbb{P}(\text{Pom} \times V_{\text{int}}) \)

\( \mathcal{P} : \text{Exp}_{\text{bool}} \rightarrow \mathbb{P}(\text{Pom} \times V_{\text{bool}}) \)

are defined by structural induction, e.g.

\( \mathcal{P}(i_\alpha) = \{ (\{i_\alpha = v\}, v) \mid v \in V_{\text{int}} \} \)

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

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

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

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

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

- The boolean expression $x = x$ has

  $\mathcal{P}(x = x)_{tt} = \{\{x=v_1\} \parallel \{x=v_2\} \mid v_1 = v_2\}$

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

We allow for interaction with environment.
Pomset Semantics for commands

\[ \mathcal{P} : \text{Com} \rightarrow \mathcal{P}(\text{Pom}) \]

is defined by structural induction:

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

The program

\[ y_{at} := 1 \parallel \text{if } y_{at} = 0 \text{ then } x := y_{at} + 1 \text{ else skip } \]

denotes the set of pomsets

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

where

\[ P_v: \quad y_{at} := 1 \quad y_{at} = 0 \quad Q_v: \quad y_{at} := 1 \quad y_{at} = v \]

\[ y_{at} = v \]

\[ x := v + 1 \]
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 \( P \in \text{loop}^* \text{stop} \cup \text{loop}^\omega \), and

\[
\begin{align*}
\text{loop} & = \{ \{ y_{at} = 0 \cdot x_{at} = v \cdot x_{at} ::= v + 1 \} \mid v \in V_{int} \} \\
\text{stop} & = \{ \{ y_{at} = v \} \mid v \neq 0 \}
\end{align*}
\]

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

What is special about the pomsets in \( y_{at} ::= 1 \parallel \text{loop}^\omega \)?
Local Resources

in more detail

\[ \mathcal{P}(\text{resource } r \text{ in } c) = \{ P \setminus r \mid P \in \mathcal{P}(c)_r \} \]

(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) \(P \setminus r\) 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:

\[
\text{lock } r \; \cdots \; \text{unlock } r \; \cdots \; \text{lock } r \; \cdots
\]

(ii) \(r\)-actions are invisible outside the scope

Only uses the pomsets of \(c\) that can be suitably extended
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

\[
\begin{align*}
x_{at} & := 1 & y_{at} & := 1 \\
\downarrow & & \downarrow \\
y_{at} & := v_1 & x_{at} & := v_2 \\
\downarrow & & \downarrow \\
z_1 & := v_1 & z_2 & := v_2
\end{align*}
\]

where \(v_1, v_2 \in V_{\text{int}}\).

▶ 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

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

- Each finite pomset of this program has the form

\[
\begin{align*}
x &: = 37 \\
y_{at} &: = 1 \\
(y_{at} &= 0)^k
\end{align*}
\]

\begin{align*}
\downarrow && \downarrow \\
y_{at} &= v \quad (v \neq 0) \\
\downarrow && \downarrow \\
x &= v' \\
\downarrow && \\
z &= v'
\end{align*}

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

\[
\begin{array}{c c c}
\text{\(x_{at} := 1\)} & \text{\(y_{at} := 1\)} & \text{\(x_{at} = v_{1}\)} & \text{\(y_{at} = v'_{1}\)} \\
\downarrow & \downarrow & \downarrow & \downarrow \\
\text{\(z_{1} := v_{1}\)} & \text{\(w_{1} := v'_{1}\)} & \text{\(y_{at} = v_{2}\)} & \text{\(x_{at} = v'_{2}\)} \\
\downarrow & \downarrow & \downarrow & \downarrow \\
\text{\(y_{at} = v_{2}\)} & \text{\(x_{at} = v'_{2}\)} & \text{\(z_{2} := v_{2}\)} & \text{\(w_{2} := v'_{2}\)} \\
\end{array}
\]

for \(v_{1}, v_{2}, v'_{1}, v'_{2} \in V_{\text{int}}\)
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

\[
\begin{align*}
  x_{at} := 1 & \quad x_{at} := 2 \\
  x_{at} = v_1 & \quad x_{at} = v'_1 \\
  z_1 := v_1 & \quad w_1 := v'_1 \\
  x_{at} = v_2 & \quad x_{at} = v'_2 \\
  z_2 := v_2 & \quad w_2 := v'_2
\end{align*}
\]

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

\[
\begin{align*}
\text{lock } r \\
\downarrow \\
x=v_1 \\
\downarrow \\
x:=v_1 + 1 \\
\downarrow \\
\text{unlock } r \\
\text{lock } r \\
\downarrow \\
x=v_2 \\
\downarrow \\
x:=v_2 + 1 \\
\downarrow \\
\text{unlock } r
\end{align*}
\]

for $v_1, v_2 \in V_{\text{int}}$. 
Concurrent Increments using a private lock

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

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

\[
\begin{align*}
lock \ r & \quad lock \ r \\
\downarrow & \quad \downarrow \\
x = v_1 & \quad x = v_2 \\
\downarrow & \quad \downarrow \\
x := v_1 + 1 & \quad x := v_2 + 1 \\
\downarrow & \quad \downarrow \\
unlock \ r & \quad unlock \ r
\end{align*}
\]

and after erasing the \( r \)-actions, we get

\[
\begin{align*}
x = v_1 & \quad x = v_2 \\
\downarrow & \quad \downarrow \\
x := v_1 + 1 & \quad x := v_2 + 1
\end{align*}
\]
Semantic equivalence

\[ \mathcal{P}(\text{resource } r \text{ in } (inc \parallel inc)) = \mathcal{P}(x:=x + 1; x:=x + 1) \]

- The pomsets of

  \text{resource } r \text{ in } (inc \parallel inc)

  are

  \[ x=v_1 \quad \text{and} \quad x=v_2 \]

  \[ x:=v_1 + 1 \quad \text{and} \quad x:=v_2 + 1 \]

  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

framework = semantics + execution

- We have a denotational semantics

\[ \mathcal{P} : \text{Com} \rightarrow \mathcal{P}(\text{Pom}) \]

defined without reference to memory model or architecture.

- We can now introduce a notion of

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.
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,

\[
\text{initial segment } = \text{ prefix } \quad \triangleleft = \text{ concatenation}
\]

- If \( P \) is linear and \( P = P_1 \triangleleft P_2 \), then \( P_1 \) and \( P_2 \) are linear and \( P = P_1; P_2 \).
- The converse also holds.

For non-linear pomsets:

- When \( P = P_1 \parallel P_2 \) we get

\[
P = P_1 \triangleleft P_2 \text{ and } P = P_2 \triangleleft P_1
\]

but \( P \neq P_1; P_2 \) and \( P \neq P_2; P_1 \).
Concurrence

We write $P_1 \text{ co } P_2$ 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 $P_1 \text{ co } P_2$ & $\text{res}(P_1) \cap \text{res}(P_2) = \{\}$. 
- $\text{res}(P)$ is the set of lock names $r$ in actions of $P$. 

Pomset Footprints

Intuition

\[ [P] \subseteq \Sigma \times \Sigma^T \]

\([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

\[
[-] : \text{Pom} \rightarrow \mathcal{P}(\Sigma \times \Sigma^\top)
\]

is the least function such that:

**Act** If \( P \) is a singleton \( \{\lambda\} \), \( [-P] = [-\lambda] \).

**Seq** If \( P = P_1 \triangleleft P_2 \), \( (\sigma_1, \tau_1) \in [-P_1], (\sigma_2, \tau_2) \in [-P_2], [\sigma_1 \uparrow \tau_1] \uparrow \sigma_2 \),
then \( (\sigma_1 \cup (\sigma_2 \setminus \text{dom} \tau_1), [\tau_1 \uparrow \tau_2]) \in [-P] \).
If \( P = P_1 \triangleleft P_2 \) and \( (\sigma, \top) \in [-P_1] \), then \( (\sigma, \top) \in [-P] \).

**Par** If \( P = P_1 \sqcup P_2 \), \( P_1 \text{ co } P_2 \), \( \text{res}(P_1) \cap \text{res}(P_2) = \{\} \),
(\(\sigma_1, \tau_1) \in [-P_1], (\sigma_2, \tau_2) \in [-P_2], \) and \( \sigma_1 \uparrow \sigma_2 \),
then \( (\sigma_1 \cup \sigma_2, [\tau_1 \uparrow \tau_2]) \in [-P] \) and \( (\sigma_1 \cup \sigma_2, [\tau_2 \uparrow \tau_1]) \in [-P] \).

**Race** If \( P = P_1 \sqcup P_2 \), \( \neg(P_1 \text{ co } P_2) \), \( \text{res}(P_1) \cap \text{res}(P_2) = \{\} \),
(\(\sigma_1, \tau_1) \in [-P_1], (\sigma_2, \tau_2) \in [-P_2], \) and \( \sigma_1 \uparrow \sigma_2 \),
then \( (\sigma_1 \cup \sigma_2, \top) \in [-P] \).
**Explanation**

**SEQ** embodies *sequential composition* of footsteps
- \((\sigma_1, \tau_1)\) can be followed by \((\sigma_2, \tau_2)\) iff \([\sigma_1 \mid \tau_1] \uparrow \sigma_2\)
- Their cumulative footprint is represented by
  \((\sigma_1 \cup (\sigma_2 \setminus \text{dom} \tau_1), [\tau_1 \mid \tau_2])\)

**Par** and **Race** enforce a form of race-detecting *concurrent composition* of footsteps
- \((\sigma_1, \tau_1)\) can be composed with \((\sigma_2, \tau_2)\) 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 \(\text{res}(P_1) \cap \text{res}(P_2) = \{\}\) 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

\[
\begin{align*}
P & = P_1 \triangleright P_2 \\
P & \neq Q_1 \triangleright Q_2
\end{align*}
\]
Sequencing

footprints

execution

\[
(x_{at}=1, y_{at}=0) 
\quad \quad 
(x_{at}=2, y_{at}=1) 
\quad \quad 
(x_{at}=2, y_{at}=2)
\]

\[
(x_{at}:=1, y_{at}:=1) \quad \quad 
(x_{at}:=2, y_{at}:=1) 
\quad \quad 
(x_{at}:=2, y_{at}:=2)
\]

\[
\begin{align*}
(x, y) & = (0, 0), \quad (2, 1) \\
(x, y) & = (2, 1), \quad (2, 2)
\end{align*}
\]

\[
(x, y) = (0, 0), \quad (2, 2)
\]

\[
\begin{align*}
(x_{at}:=1, y_{at}:=1) & \quad \quad 
(x_{at}:=2, y_{at}:=2)
\end{align*}
\]

\[
P = P_1 \triangleright P_2
\]
Concurrence

\[ \begin{array}{c|c}
\text{P}_1 & \text{P}_2 \\
\hline
\begin{array}{cc}
x_{at} := 1 & y_{at} := 0 \\
\end{array} & \begin{array}{cc}
y_{at} := 1 & x_{at} := 0 \\
\end{array} \\
\hline
\begin{array}{c}
x_{at} := 1 \\
\end{array} & \begin{array}{c}
y_{at} := 1 \\
\end{array}
\end{array} \]

\[ \text{P}_1 \perp \text{P}_2 \]

\[ ([x:0, y:0], [x:1]) \in \llbracket \text{P}_1 \rrbracket \]

\[ ([x:0, y:0], [y:1]) \in \llbracket \text{P}_2 \rrbracket \]

\[ ([x:0, y:0], [x:1, y:1]) \in \llbracket \text{P}_1 \uplus \text{P}_2 \rrbracket \]

**footprints**

**execution**

\[ \text{P} = \text{P}_1 \uplus \text{P}_2 \]
Composing Footprints

### Sequential

\[
(\sigma_1, \tau_1) \in \llbracket P_1 \rrbracket \\
\ld\dfrac{P = P_1 \triangledown P_2 \; (\sigma_2, \tau_2) \in \llbracket P_2 \rrbracket \; [\sigma_1|\tau_1] \uparrow \sigma_2}{(\sigma_1 \cup \sigma_2 \setminus \tau_1, [\tau_1|\tau_2]) \in \llbracket P \rrbracket}
\]

### Concurrent

\[
(\sigma_1, \tau_1) \in \llbracket P_1 \rrbracket \\
\ld\dfrac{P_1 \perp P_2 \; (\sigma_2, \tau_2) \in \llbracket P_2 \rrbracket \; \sigma_1 \uparrow \sigma_2}{(\sigma_1 \cup \sigma_2, [\tau_2|\tau_1]) \in \llbracket P_1 \cup P_2 \rrbracket \; (\sigma_1 \cup \sigma_2, [\tau_1|\tau_2]) \in \llbracket P_1 \cup P_2 \rrbracket}
\]
Examples

- The pomset

\[
x_{at} := 1 \quad y_{at} := 1
\]
\[
y_{at} = 0 \quad x_{at} = 0
\]

has footprint

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

- The pomset

\[
x_{at} := 1 \quad x_{at} := 2
\]
\[
x_{at} = 2
\]

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

\[ \{P_v \mid v \in V_{\text{int}}\} \cup \{Q_v \mid v \neq 0\}, \]

where

\[ P_v : y_{at} := 1 \quad y_{at} = 0 \quad Q_v : y_{at} := 1 \quad y_{at} = v \]

\[ \begin{array}{c}
\downarrow \\
y_{at} = v \\
\downarrow \\
x := v + 1
\end{array} \]

\[ \begin{array}{c}
\downarrow \\
y_{at} = v \\
\downarrow \\
\delta
\end{array} \]

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:
\[ \mathcal{E}(P) = \{ (\sigma, [\sigma | \tau_1]) \mid \exists \sigma_1 \subseteq \sigma. (\sigma_1, \tau_1) \in \llbracket P \rrbracket \}, \]
where we let \([\sigma | \top] = \top\).

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

Properties

Justification

▶ Each execution \((\sigma, \sigma') \in \mathcal{E}(P)\) is “justified”
  by a footprint \((\sigma_1, \tau_1) \in \llbracket P \rrbracket\), such that \(\sigma_1 \subseteq \sigma \) & \(\sigma' = [\sigma \mid \tau_1]\).
▶ Footprints are derived using \texttt{Act, Seq, Par, Race}.

Sequencing

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

Concurrence

▶ If \(P = P_1 \parallel P_2\), \((\sigma, \sigma_1) \in \mathcal{E}(P_1)\) 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, λ₁ must get executed before λ₂.*

- All state changes come from actions of P:
  
  If \((σ, σ') ∈ \mathcal{E}(P)\) and \(σ'(i) = v' ≠ σ(i)\),
  there is an occurrence of \(i_α := v\) in \(P\).

- Hence, **no out-of-thin-air writes.**
Out-of-thin-air

In some weak memory model specifications the execution graph

\[ y = 42 \quad x = 42 \]
\[ x := 42 \quad y := 42 \]

(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.
Theorem

Every (finite) execution \((σ, σ') ∈ E(P)\) is expressible as a sequence of phases

\[
σ = σ_0 \xrightarrow{P_0} σ_1 \xrightarrow{P_1} σ_2 \cdots \xrightarrow{P_n} σ_{n+1} = σ'
\]

where \(P = P_0 ◁ P_1 \cdots ◁ P_n\) and each phase is justified by footprint, i.e. for each \(j\), \(∃(τ_j, τ'_j) ∈ \llbracket P_j \rrbracket\) with \(τ_j ⊆ σ_j\) and \(σ_{j+1} = [σ_j | τ'_j]\).

- 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 \(\text{Act}, \text{Seq}, \text{Par}, \text{Race}\) 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**

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

$$P = P_0 \triangleleft P_1 \ldots \triangleleft P_n \ldots$$

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

$$\sigma = \sigma_0 \xrightarrow{P_0} \sigma_1 \xrightarrow{P_1} \sigma_2 \ldots \sigma_n \xrightarrow{P_n} \sigma_{n+1} \ldots$$

*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

\[
\begin{align*}
\text{[resource } r \text{ in } (inc \mid inc)] &= \{([x : v], [x : v + 2]) \mid v \in V_{int} \} \\
\mathcal{E}(\text{resource } r \text{ in } (inc \mid inc)) &= \{ (\sigma, [\sigma \mid x : v + 2]) \mid x : v \in \sigma \}
\end{align*}
\]
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 $\sigma$.
- Extract the final state $\sigma'$.

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

<table>
<thead>
<tr>
<th>Litmus Test</th>
<th>SC</th>
<th>TSO</th>
<th>rel/acq</th>
<th>$\mathcal{E}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Store Buffering</td>
<td>$\times$</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>2. Message Passing</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>3. IRIW</td>
<td>$\times$</td>
<td>$\times$</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>4. Coherence</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>5. Optimization</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
Litmus Test 1

Store Buffering

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

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

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

In this execution the reads see stale values.
Litmus Test 2
Message Passing

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

- Each finite pomset has the form

```
\begin{array}{c}
x:=37 \\
\downarrow \\
y_{at}:=1 \\
\downarrow \\
y_{at}=v \quad (v \neq 0) \\
\downarrow \\
x=v' \\
\downarrow \\
z:=v'
\end{array}
```

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

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

- The pomset

\[
\begin{align*}
x_{at} &= 1 & y_{at} &= 1 \\
x_{at} &= 1 & y_{at} &= 1 \\
z_1 &= 1 & w_1 &= 1 \\
y_{at} &= 0 & x_{at} &= 0 \\
z_2 &= 0 & w_2 &= 0
\end{align*}
\]

is executable from \([x : 0, y : 0, \ldots]\)

- Threads see the writes to \(x\) and \(y\) in different orders.
The pomset

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

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 $\textit{inc}$ be \texttt{lock } $r$; $x:=x+1$; \texttt{unlock } $r$.

- The pomsets of

$$\text{resource } r \text{ in } (\textit{inc} \parallel \textit{inc})$$

have form

$$x=v_1 \quad \quad \quad \quad x=v_2$$

$$\quad \downarrow \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \downarrow$$

$$x:=v_1 + 1 \quad \quad \quad x:=v_2 + 1$$

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

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

$$\mathcal{E}(\text{resource } r \text{ in } (\textit{inc} \parallel \textit{inc})) = \mathcal{E}(x:=x+2)$$
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 $\llbracket c_1 \rrbracket = \llbracket c_2 \rrbracket$, then $\mathcal{E}(C[c_1]) = \mathcal{E}(C[c_2])$.

**Examples**

\[
\begin{align*}
\mathcal{E}(c \parallel \text{lock } r; x:=x+1; x:=x+1; \text{unlock } r) &= \mathcal{E}(c \parallel \text{lock } r; x:=x+2; \text{unlock } r) \\
\mathcal{E}(c \parallel c_{11}; x:=1; y:=2; c_{12}) &= \mathcal{E}(c \parallel c_{21}; y:=2; x:=1; c_{22})
\end{align*}
\]
Semantic Properties

- $\mathcal{P}$ is *compositional*
  - supports syntax-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 $\text{inc}^{(n)}$ be $\text{inc} \parallel \cdots \parallel \text{inc}$ (n times).

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

\[
\begin{align*}
    c_1 \| (c_2 \| c_3) & \equiv_P (c_1 \| c_2) \| c_3 \\
    c_1 \| c_2 & \equiv_P c_2 \| c_1 \\
    c_1; (c_2; c_3) & \equiv_P (c_1; c_2); c_3 \\
    c; \text{skip} & \equiv_P \text{skip}; c \equiv_P c \\
    c \| \text{skip} & \equiv_P c
\end{align*}
\]

and *scope contraction*

\[
\text{resource } r \text{ in } (c_1 \| c_2) \equiv_P (\text{resource } r \text{ in } c_1) \| c_2
\]
when \( r \) 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
  \[ c_1 \parallel (c_2 \parallel c_3) \equiv_{\mathcal{E}} (c_1 \parallel c_2) \parallel c_3 \]
  and *scope contraction*

  \[ \text{resource } r \text{ in } (c_1 \parallel c_2) \equiv_{\mathcal{E}} \text{(resource } r \text{ in } c_1) \parallel c_2 \]
  when \( r \) 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 \((\sigma, \sigma')\).
- 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
  
  \textit{happens-before, reads-from, modification-order}

  We just use program order $<$ and can \textit{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 \( \text{PAR} \)

\[ \mathcal{E}(P) \neq \bigcup \{ \mathcal{E}(P') \mid P' \in \text{LIN}(P) \} \]

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

- **Brookes Is Relaxed, Almost!**
  - 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

\[P = P_0 \bowtie P_1 \bowtie \cdots \bowtie P_n\]

and

\[\sigma_0 \xrightarrow{P_0} \sigma'_0 \& \sigma_1 \xrightarrow{P_1} \sigma'_1 \cdots \& \sigma_n \xrightarrow{P_n} \sigma'_n\]

- 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

Progress is impossible without change, and those who cannot change their minds cannot change anything.

- George Bernard Shaw
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.