A DENOTATIONAL APPROACH TO RELEASE/ACQUIRE CONCURRENCY
GOAL

Design a standard, monad-based denotational semantics (Moggi [1991])

Using Brookes-style [1996], totally-ordered traces

For weak, shared-memory model

RELEASE/ACQUIRE
WHY RELEASE/ACQUIRE?

RA is an important fragment of C/C++, enables decentralized architectures (POWER)

First adaptation of Brookes’s traces to a software model (compositional parallelism)

Intricate causal semantics, not overwhelmingly detailed

Threads can disagree about the order of writes (non-multi-copy-atomic)

Supports flag-based synchronization (e.g. for signaling a data structure is ready)

Supports important transformations (e.g. thread sequencing, write-read-reorder)

Supports read-modify-write atomicity
TRACE-BASED SEMANTICS

Brookes [1996]

Main ingredient: linearly-ordered traces of state-transitions that sequence and interleave

\[
\langle \mu_1, q_1 \rangle \langle \mu_2, q_2 \rangle \cdots \langle \mu_{n-1}, q_{n-1} \rangle \langle \mu_n, q_n \rangle
\]
TRACE-BASED SEMANTICS

Brookes [1996]

Main ingredient: linearly-ordered traces of state-transitions that sequence and interleave

\[ \langle \mu_1, \varrho_1 \rangle \langle \mu_2, \varrho_2 \rangle \cdots \langle \mu_{n-1}, \varrho_{n-1} \rangle \langle \mu_n, \varrho_n \rangle \]

\[ \langle \mu_1', \varrho_1' \rangle \langle \mu_2', \varrho_2' \rangle \cdots \langle \mu_n', \varrho_n' \rangle \]

\[ \langle \varrho_1, \varrho_1' \rangle \langle \varrho_2, \varrho_2' \rangle \cdots \langle \varrho_n, \varrho_n' \rangle \]
TRACE-BASED SEMANTICS
Brookes [1996]

Main ingredient: linearly-ordered traces of state-transitions that sequence and interleave

\[
\langle \mu_1, q_1 \rangle \langle \mu_2, q_2 \rangle \cdots \langle \mu_{n-1}, q_{n-1} \rangle \langle \mu_n, q_n \rangle
\]

\[
\langle \mu_1, \mu'_1 \rangle \langle \mu_2, \mu'_2 \rangle \cdots \langle \mu_n, \mu'_n \rangle \langle q_1, q'_1 \rangle \langle q_2, q'_2 \rangle \cdots \langle q_n, q'_n \rangle
\]
TRACE-BASED SEMANTICS

Brookes [1996]

Main ingredient: linearly-ordered traces of state-transitions that sequence and interleave

\[ \langle \mu_1, \varrho_1 \rangle \langle \mu_2, \varrho_2 \rangle \ldots \langle \mu_{n-1}, \varrho_{n-1} \rangle \langle \mu_n, \varrho_n \rangle \]

\[ \langle \varrho_1, \varrho'_1 \rangle \langle \mu_1, \mu'_1 \rangle \langle \mu_2, \mu'_2 \rangle \langle \varrho_2, \varrho'_2 \rangle \ldots \langle \mu_n, \mu'_n \rangle \langle \varrho_n, \varrho'_n \rangle \]

INTERLEAVE
TRACE-BASED SEMANTICS

Brookes [1996]
- Denotational semantics $\parallel \rightarrow \parallel$ for concurrency
- Idealized model - Sequential Consistency (SC)
- Follows operational semantics

Jagadeesan, Petri, Riely [2012]
- Adapts traces to TSO (hardware model)
- Follows operational semantics too
  - Relatively close to SC

Main ingredient: linearly-ordered traces of state-transitions that sequence and interleave

$\langle \mu_1, \psi_1 \rangle \langle \mu_2, \psi_2 \rangle \cdots \langle \mu_{n-1}, \psi_{n-1} \rangle \langle \mu_n, \psi_n \rangle$

This work
- Adapts traces to RA (software model)
- Kang et al. [2017] operational presentation
  - Much more complex notion of state
CONTRIBUTION

Directionally Adequate denotational semantics for RA based on linearly-ordered traces

Standard (CbV) semantics [Moggi 1991] enables structural transformations (e.g. \( \| K \; (M; N) \| = \| (K; M); N \| \)) has higher-order functions for free etc.

Abstract enough to justify every transformation discussed in the literature that we know of (but no full-abstraction)

New challenge — non-operational interpretation: each trace represents a possible behavior as a Rely/Guarantee sequence
RELEASE/ACQUIRE
TYPICAL EXAMPLES

Store Buffering

\[
\begin{align*}
x & := 0; y := 0; \\
x & := 1; \quad y := 1; \\
y ? & \quad | \quad x ?
\end{align*}
\]

Message Passing

\[
\begin{align*}
x & := 0; y := 0; \\
x & := 1; \quad y ?; \\
y & := 1 \quad | \quad x ?
\end{align*}
\]
## TYPICAL EXAMPLES

<table>
<thead>
<tr>
<th>Store Buffering</th>
<th>Message Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 0; y := 0; )</td>
<td>( x := 0; y := 0; )</td>
</tr>
<tr>
<td>( x := 1; )</td>
<td>( x := 1; )</td>
</tr>
<tr>
<td>( y := 1; )</td>
<td>( y ? )</td>
</tr>
<tr>
<td>( y ? ) \quad //0</td>
<td>( y ? ) \quad //0</td>
</tr>
<tr>
<td>( x ? ) \quad //0</td>
<td>( x ? ) \quad //0</td>
</tr>
</tbody>
</table>
TYPICAL EXAMPLES

**Store Buffering**

\[ x := 0; y := 0; \]
\[ x := 1; \]
\[ y := 1; \]
\[ y \Leftarrow \text{0} \]
\[ x \Leftarrow \text{0} \]

**Message Passing**

\[ x := 0; y := 0; \]
\[ x := 1; \]
\[ y \Leftarrow \text{?} \]
\[ y := 1 \]
\[ x \Leftarrow \text{?} \]
TYPICAL EXAMPLES

Store Buffering

\[
\begin{align*}
  x &:= 0; y := 0; \\
  x &:= 1; \quad y := 1; \\
  y? &\quad //0 \quad x? &\quad //0
\end{align*}
\]

Message Passing

\[
\begin{align*}
  x &:= 0; y := 0; \\
  x &:= 1; \quad y?; \\
  y &:= 1 \quad //0 \quad x?
\end{align*}
\]

Propagation is not instant
TYPICAL EXAMPLES

Store Buffering

\[
\begin{align*}
x &:= 0; y := 0; \\
x &:= 1; \\
y &? /0 \\
x &? /0
\end{align*}
\]

Message Passing

\[
\begin{align*}
x &:= 0; y := 0; \\
x &:= 1; \\
y &?; /1 \\
y := 1 \\
x &? /0
\end{align*}
\]

Propagation is not instant
TYPICAL EXAMPLES

Store Buffering

\[ x := 0; y := 0; \]
\[ x := 1; \]
\[ y := 1; \]
\[ y? \quad // 0 \]
\[ x? \quad // 0 \]

Message Passing

\[ x := 0; y := 0; \]
\[ x := 1; \]
\[ y?; \quad // 1 \]
\[ y := 1 \]
\[ x? \quad // 0 \]
TYPICAL EXAMPLES

Store Buffering

\[ x := 0; y := 0; \]
\[ x := 1; y := 1; \]
\[ y \not= 0 \]
\[ x ? \not= 0 \]

Message Passing

\[ x := 0; y := 0; \]
\[ x := 1; \]
\[ y ? ; // 1 \]
\[ y := 1 \]
\[ x ? ; // 0 \]

Propagation is not instant

Propagation respects causality
RELEASE/ACQUIRE VIEW-BASED OPERATIONAL SEMANTICS

Kang et al. [2017]

- **Memory**: Timeline per location (e.g. x, y, z)
- Populated with immutable messages (e.g. x0, y0, z0)
- Each thread’s view points to a msgs on each timeline (e.g. T1)
- Thread’s cannot read from “the past”
- Each msg’s view points to a msg on each other timelines (e.g. y1)
- Message views are used for enforcing causal propagation
RELEASE/ACQUIRE VIEW-BASED OPERATIONAL SEMANTICS
Kang et al. [2017]

- **Memory**: Timeline per location (e.g. x, y, z)
- Populated with immutable messages (e.g. x0, y0, z0)
- Each thread’s view points to a msgs on each timeline (e.g. T1)
- Thread’s cannot read from “the past”
- Each msg’s view points to a msg on each other timelines (e.g. y1)
- Message views are used for enforcing causal propagation
RELEASE/ACQUIRE VIEW-BASED OPERATIONAL SEMANTICS
Kang et al. [2017]

★ **Memory**: Timeline per location (e.g. x, y, z)
★ Populated with immutable messages (e.g. x₀, y₀, z₀)
★ Each thread’s view points to a msgs on each timeline (e.g. T₁)
★ Thread’s cannot read from “the past”
★ Each msg’s view points to a msg on each other timelines (e.g. y₁)
★ Message views are used for enforcing causal propagation
Memory: Timeline per location (e.g. x, y, z)
- Populated with immutable messages (e.g. x0, y0, z0)
- Each thread’s view points to a msgs on each timeline (e.g. T1)
- Thread’s cannot read from “the past”
- Each msg’s view points to a msg on each other timelines (e.g. y1)
- Message views are used for enforcing causal propagation
RELEASE/ACQUIRE VIEW-BASED OPERATIONAL SEMANTICS

Kang et al. [2017]

- **Memory:** Timeline per location (e.g. x, y, z)
- Populated with immutable messages (e.g. x0, y0, z0)
- Each thread’s view points to a msgs on each timeline (e.g. T1)
- Thread’s cannot read from “the past”
- Each msg’s view points to a msg on each other timelines (e.g. y1)
- Message views are used for enforcing causal propagation
When writing, the message:

- must be placed after thread’s view
- may be placed before others
- copies thread’s view
When writing, the message:

- must be placed after thread’s view
- may be placed before others
- copies thread’s view
must be placed after thread’s view
may be placed before others
copies thread’s view

When writing, the message:

$x := x2$
When reading, the message:

- cannot be before thread's view
- may be before others

and the thread:

- inherits the copy of the view
When reading, the message:

- cannot be before thread’s view
- may be before others

and the thread:

- inherits the copy of the view
CAUSALITY AND COMPOSITION

With first class parallelism

\[ L \parallel \left( T; \left( (U; M; D) \parallel R \right); B \right) \]

Generalized Sequencing

\((M_1; M_2) \parallel (K_1; K_2) \rightarrow (M_1 \parallel K_1); (M_2 \parallel K_2)\)
TRACE-BASED SEMANTICS IN RA

Terms denote sets of traces $\mathcal{M} \ni \tau$

Each trace represents a possible behavior as a Rely/Guarantee sequence

$\alpha \langle \mu_1, q_1 \rangle \langle \mu_2, q_2 \rangle \ldots \langle \mu_{n-1}, q_{n-1} \rangle \langle \mu_n, q_n \rangle \omega :: r$

Initial View Sequence of Transitions Final View Returns
TRACE-BASED SEMANTICS IN RA

Initial View

Sequence of Transitions

Final View

Returns

Before or ||

Rely On $\mu_1$ To Guarantee $\varrho_1$

Then

Rely On $\mu_2$ To Guarantee $\varrho_2$

Then...

Guarantee to the sequential environment to return $r$
TRACE-BASED SEMANTICS IN RA

Before

\[ \alpha \langle \mu_1, \varrho_1 \rangle \langle \mu_2, \varrho_2 \rangle \cdots \langle \mu_{n-1}, \varrho_{n-1} \rangle \langle \mu_n, \varrho_n \rangle \omega : r \]

After

Rely on the sequential environment to reveal messages before \( \alpha \)

Guarantee to the sequential environment to reveal messages before \( \omega \)

Initial View

Sequence of Transitions

Final View

Returns
Analogous to Brookes’s

**Stutter**

\[ \alpha \xi \eta \omega \therefore r \in \left\lfloor M \right\rfloor \]

\[ \alpha \xi \langle \mu, \mu \rangle \eta \omega \therefore r \in \left\lfloor M \right\rfloor \]

*Propagate Reliance as a Guarantee*

**Mumble**

\[ \alpha \xi \langle \mu, \rho \rangle \langle \rho, \theta \rangle \eta \omega \therefore r \in \left\lfloor M \right\rfloor \]

\[ \alpha \xi \langle \mu, \theta \rangle \eta \omega \therefore r \in \left\lfloor M \right\rfloor \]

*Rely on an omitted Guarantee*
View Closures

Rewind

\[ \alpha' \leq \alpha \]
\[ \alpha \xi \omega : r \in \| M \| \]

\[ \alpha' \xi \omega : r \in \| M \| \]

Relying on more being revealed

Forward

\[ \alpha \xi \omega : r \in \| M \| \]
\[ \omega \leq \omega' \]

\[ \alpha \xi \omega' : r \in \| M \| \]

Guaranteeing less being revealed
**COMPOSITION**

**Sequential**

\[
\alpha \xi_1 \kappa \therefore r_1 \in \| M_1 \| \quad \kappa \xi_2 \omega \therefore r_2 \in \| M_2 \| [x \mapsto r_1]
\]

\[
\alpha \xi_1 \xi_2 \omega \therefore r_2 \in \| \text{let } x = M_1 \text{ in } M_2 \|
\]

**Parallel**

\[
\forall i \in \{1,2\} . \quad \alpha \xi_i \omega \therefore r_i \in \| M_i \|
\]

\[
\alpha \xi \omega \therefore \langle r_1, r_2 \rangle \in \| M_1 \parallel M_2 \|
\]
ABSTRACTION
WHAT WE CAN JUSTIFY

with Stutter, Mumble, Rewind, and Forward

Structural equivalences, e.g. if $K$ is effect-free then
\[
\begin{align*}
\forces \text{if } K \text{ then } M; P_1 \text{ else } M; P_2 \forces \text{if } K \text{ then } P_1 \text{ else } P_2
\end{align*}
\]

Laws of Parallel Programming, e.g. Generalized Sequencing
\[
\begin{align*}
\forces (M_1; M_2) || (K_1; K_2) \supseteq \forces (M_1 || K_1); (M_2 || K_2)
\end{align*}
\]

Some memory access related transformations, e.g. Read-Read Elimination
\[
\begin{align*}
\forces \text{let } a = x? \text{ in } \text{let } b = x? \text{ in } \langle a, b \rangle \supseteq \forces \text{let } c = x? \text{ in } \langle c, c \rangle
\end{align*}
\]
SEMANTIC INVARIANTS ON TRACES

Read Elimination

$x?; M \rightarrow M$

operational invariant becomes denotational requirement
views point to messages that carry a smaller view

$$\kappa \langle \mu, \mu \rangle \kappa \vdash \langle \rangle \in \ll \langle \rangle \rr \implies \exists v. \kappa \langle \mu, \mu \rangle \kappa \vdash v \in \ll x? \rr$$
Some transformations are valid even without preserving state

Traces cannot strictly correspond to operational semantics (e.g. Transition ≡ exec. steps)

\[ \alpha \langle \mu_1, \varrho_1 \rangle \langle \mu_2, \varrho_2 \rangle \cdots \langle \mu_{n-1}, \varrho_{n-1} \rangle \langle \mu_n, \varrho_n \rangle \omega : r \]

\[ \cdots \langle \mu_2, - \rangle, M_1 \rightarrow^* \langle \rho_2, - \rangle, M_2 \cdots \]
ABSTRACT CLOSURES

- **Absorb** a redundant local message into a following one
  (e.g. $\| x := 0; x := 1 \| \supseteq \| x := 1 \|$)

- **Dilute** a message by a redundant local message
  (e.g. $\| x ? \| \supseteq \| \text{FAA}[x](0) \|$)

- **Tighten** the encumbering view that a local message carries
  (e.g. $\| x := 1; y ? \| \supseteq \| (x := 1 || y ?).\text{snd} \|$)

**Rewrite**

\[
\pi \in \| M \| \quad \pi \rightarrow \tau
\]
\[
\tau \in \| M \|
\]
ABSTRACT REWRITE RULES

Write-Read Deorder + LoPP + Struct $\Rightarrow$ Write-Read Reorder

\[ \| x := 1; y? \| \supseteq \| (x := 1 \parallel y?).\text{snd} \| \]

GUARANTEE IS WEAKER
BECAUSE LOADING THIS MESSAGE OBSCURES MORE
Because traces are not operational, the adequacy proof is more nuanced:

- We define a similar denotational semantics $\llbracket M \rrbracket$ but without the abstract rules
- We show it is adequate (easier because it has an operational interpretation)
- We show $\llbracket M \rrbracket = \llbracket M \rrbracket^\dagger$ — it is enough to apply the closure on top
- We show that the abstract closures preserve observations
Laws of Parallel Programming

Symmetry \( M \parallel N \rightarrow \text{match } N \parallel M \text{ with } (y, x). \langle x, y \rangle \)

Generalized Sequencing
\[(\text{let } x = M_1 \text{ in } M_2) \parallel (\text{let } y = N_1 \text{ in } N_2) \rightarrow \text{match } M_1 \parallel N_1 \text{ with } (x, y). M_2 \parallel N_2\]

Eliminations

Irrelevant Read \( \ell? \; \langle \rangle \rightarrow \langle \rangle \)

Write-Write \( \ell := v \; \ell := w \overset{\text{Ab}}{\rightarrow} \ell := w \)

Write-Read \( \ell := v \; \ell? \rightarrow \ell := v \; v \)

Write-FAA \( \ell := v \; \text{FAA}(\ell, w) \overset{\text{Ab}}{\rightarrow} \ell := (v + w) \; v \)

Read-Write \( \text{let } x = \ell? \text{ in } \ell := (x + v) \; x \rightarrow \text{FAA}(\ell, v) \)

Read-Read \( \langle \ell?, \ell? \rangle \rightarrow \text{let } x = \ell? \text{ in } \langle x, x \rangle \)

Read-FAA \( \langle \ell?, \text{FAA}(\ell, v) \rangle \rightarrow \text{let } x = \text{FAA}(\ell, v) \text{ in } \langle x, x \rangle \)

FAA-Read \( \langle \text{FAA}(\ell, v), \ell? \rangle \rightarrow \text{let } x = \text{FAA}(\ell, v) \text{ in } \langle x, x + v \rangle \)

FAA-FAA \( \langle \text{FAA}(\ell, v), \text{FAA}(\ell, w) \rangle \overset{\text{Ab}}{\rightarrow} \text{let } x = \text{FAA}(\ell, v + w) \text{ in } \langle x, x + v \rangle \)

Others

Irrelevant Read Introduction \( \langle \rangle \rightarrow \ell? \; \langle \rangle \)

Read to FAA \( \ell? \overset{\text{Di}}{\rightarrow} \text{FAA}(\ell, 0) \)

Write-Read Deorder \( \langle (\ell := v), \ell'? \rangle \overset{T_1}{\rightarrow} (\ell := v) \parallel \ell'? \quad (\ell \neq \ell') \)

Write-Read Reorder \( \langle (\ell := v), \ell'? \rangle \overset{T_1}{\rightarrow} \text{let } x = \ell'? \text{ in } (\ell := v) \; x \quad (\ell \neq \ell') \)
CONCLUSION
CONCLUSION

- Standard, adequate and fully-compositional denotational semantic for RA
- More nuanced traces
- Sufficiently abstract: validates all RA transformations that we know of (memory access, laws of parallel programming, structural transformations)
- Extended RA view-based machine with compositional (i.e. first-class) parallelism (weak-memory models are usually studied with top-level parallelism)
LIMITATIONS

- Parsimonious in features (e.g. no recursion)
- No type-and-effect system
- No algebraic presentation
- No non-atomics, not the full C/C++ model
- No full abstraction theorem even for first-order
FUTURE DIRECTIONS

Address the mentioned limitations, e.g. promising semantics to cover more of C/C++

Algebraic effects as Rely/Guarantee traces

\[
\begin{align*}
|\emptyset| & : \text{Term}_{\{L, U\}} X \rightarrow \mathcal{P}_{\text{fin}} (\mathbb{T}X) \\
|x| & := \{\langle \rangle \ll x\} \\
|L_\ell \langle t_v \rangle_{v \in \text{Val}}| & := \bigcup_{v \in \text{Val}} \{(R_\ell, v :: t) \ll x \mid t :: x \in \{t_v\}\} \\
|U_\ell, v t| & := \{(G_\ell, v :: t) \ll x \mid t :: x \in \{t\}\}
\end{align*}
\]
OPERATIONAL SEMANTICS

1ST-CLASS PARALLELISM

ABSTRACT CLOSURES

ADEQUACY PROOF

RELY/GUARANTEE TRACES
**REWRITE RULE: ABSORB**

**Write Eliminations**

\[
x := 0; \ x := 1 \implies x := 1
\]

\[
x := 0; \ CAS[x](0,1) \implies x := 1
\]

Eliminate redundant message
Write Eliminations

\[ x? \rightarrow CAS[x](1,1) \]

\[ CAS[x](1,1) \rightarrow FAA[x](0) \]