A New Semantics for ABS

Reiner Hähnle
Department of Computer Science, TU Darmstadt
Joint Work with Crystal C. Din, Einar B. Johnsen, Ka I Pun, S. Lizeth T. Tarifa

International ABS Workshop 2018, TU Darmstadt
[Invited Paper at Tableaux 2017, LNCS 10501, pp. 22–43]
Once Upon a Time in Darmstadt
Once Upon a Time in Darmstadt

One fine evening . . .

we decided to develop a new sequent calculus for a program logic for ABS
Would be Good to Have a Formal Semantics . . .

. . . to guide the design of the calculus’ rules
. . . to prove soundness and, eventually, completeness
Semantics

Would be Good to Have a Formal Semantics ...

... to guide the design of the calculus’ rules
... to prove soundness and, eventually, completeness

Wanted: Trace Semantics for a Concurrent, Distributed Language

Given a program statement $s$ and an initial state $\sigma$, the semantics of $s$ is the set of all possible execution traces $\tau$ that are possible when $s$ is started in $\sigma$

- **Trace**: a possibly empty, possibly infinite, sequence of execution states
- **Global semantics**: state holds set of processors, each with own heap
- **Concurrent behavior dependent on scheduler**: set of traces
Reduction rules pattern match on runtime configurations \sim states

Typical Reduction Rule: object creation:

\[
\begin{align*}
    n(b, o, \sigma, T \; z = \textbf{new} \; C(v); s) & \Rightarrow b'(\top) \parallel n'(b', o', \sigma'_{\text{init}}, s_{\text{task}}) \parallel o'[b', c, \sigma_{\text{init}}] \parallel n(b, o, \sigma, s\{z/o'\})
\end{align*}
\]

where \( b', o', n' \) new; \( T \; f; s' \) init block of \( C; \sigma_{\text{init}} = T \; f; s_{\text{task}} = s'\{\textbf{this}/o'; \textbf{suspend}\} \)
A Clash

What we Have

Programming Language: Structural operational semantics (SOS)

What we Need

Sequent Calculus: Model theoretic, denotational semantics
What’s Wrong with SOS?

SOS Rules Define Interpreter of Target Language

- Many rules, often 3–5 for each statement (> 60 for ABS)
- Not modular:
  - No separation between local and global computation
  - Next applicable rule depends on current configuration
  - Small rule modifications have unforeseeable consequences
What’s Wrong with SOS?

**SOS Rules Define Interpreter of Target Language**

- Many rules, often 3–5 for each statement (≥ 60 for ABS)
- Not modular:
  - No separation between local and global computation
  - Next applicable rule depends on current configuration
  - Small rule modifications have unforeseeable consequences

**Program Logics Based on (forward / backward) Symbolic Execution**

- Ill-matched to SOS
- Better: denotational semantics with model theoretic flavor
Ok, We Need a Denotational Semantics
—So What?

There is none! A least not for a complex, concurrent programming language KeY, Why, Dafny, but even Dijkstra, Hoare Calculus only C, C++, Java, ABS SOS only Concurrent separation logic Stephen Brookes' action traces

Starting Point

Model theoretic semantics for while language in:
Ok, We Need a Denotational Semantics
—So What?

There is none! A least not for a complex, concurrent programming language
Ok, We Need a Denotational Semantics
—So What?

There is none! A least not for a complex, concurrent programming language

KeY, Why, Dafny, but even Dijkstra, Hoare  Calculus only
C, C++, Java, ABS  SOS only
Concurrent separation logic  Stephen Brookes’ action traces
Ok, We Need a Denotational Semantics —So What?

There is none! At least not for a complex, concurrent programming language.

KeY, Why, Dafny, but even Dijkstra, Hoare Calculus only
C, C++, Java, ABS SOS only
Concurrent separation logic Stephen Brookes’ action traces

Starting Point

Model theoretic semantics for while language in:
Richard Bubel, Crystal Chang Din, Reiner Hähnle, Keiko Nakata.
A Dynamic Logic with Traces and Coinduction. TABLEAUX 2015: 307-322
Guest this

begin execution of eat

Fut<Meal> fm = w!order(d)

order(d) on w

deal

prepare(d) on c

begin execution of prepare(d)

suspension

get fc

termination

await fc?

get fc

termination

await fm?
ABS Communication Structure

**Guest this**

**begin execution of eat**

**Fut<Meal> fm = w!order(d)**

**begin execution of order(d)**

**Fut<Meal> fc = c!prepare(d)**

**delay**
begin execution of eat

Fut<Meal> fm = w!order(d)

begin execution of order(d)

Fut<Meal> fc = c!prepare(d)

begin execution of prepare(d)

delay
begin execution of eat
\text{Fut}<\text{Meal}> fm = w!\text{order}(d)

begin execution of order(d)
\text{Fut}<\text{Meal}> fc = c!\text{prepare}(d)

begin execution of prepare(d)
\text{await} fc?

termination
begin execution of eat
Fut<Meal> fm = w!order(d)

begin execution of order(d)
Fut<Meal> fc = c!prepare(d)

begin execution of prepare(d)
await fc?
get fc

delay

suspension
ABS Communication Structure

```
Guest this
begin execution of eat
Fut<Meal> fm = w!order(d)

begin execution of order(d)
Fut<Meal> fc = c!prepare(d)

begin execution of prepare(d)

await fc?
get fc
termination

await fm?
delay
suspension

termination

prepare(d) on c

order(d) on w
```
**ABS Communication Structure**

- **Guest this**
  - Begin execution of `eat`
  - `Fut<Meal> fm = w!order(d)`

- **order(d) on w**
  - Begin execution of `order(d)`
  - `Fut<Meal> fc = c!prepare(d)`

- **prepare(d) on c**
  - Begin execution of `prepare(d)`
  - `await fc?`
  - `termination`
  - `get fc`  
  - `termination`

- **delay**
  - `await fm?`

- **suspension**
  - `get fm`
Given an ABS statement, compute all possible finite or infinite traces

... in the following manner:

Local — for given initial state, current object this, heap, future destiny

Modular — evaluation of one statement does not depend on others’

Composable — obtain global behavior by composition of object-local behavior
A Denotational Semantics for ABS

Given an ABS statement, compute all possible finite or infinite traces... in the following manner:

- **Local** — for given initial state, current object
- **Modular** — evaluation of one statement does not depend on others’
- **Composable** — obtain global behavior by composition of object-local behavior

Behavior depends on whether \( f, f' \) resolved (starvation, blocking!)

Conditional depends on value \( v \) from previous asynchronous call

Execution of \( m \) requires value of \( o \)
A Denotational Semantics for ABS

Given an ABS statement, compute all possible finite or infinite traces
... in the following manner:

- **Local** — for given initial state, current object this, heap, future destiny
- **Modular** — evaluation of one statement does not depend on others’
- **Composable** — obtain global behavior by composition of object-local behavior

```plaintext
v = f\_get; \text{ if } (v == 0) \text{ then } o.m() \text{ else } \text{await } f'?
```

- Behavior depends on whether $f, f'$ resolved (starvation, blocking!)
- Conditional depends on value $v$ from previous asynchronous call
- Execution of $m$ requires value of $o$
Despair? Sit Down and Cry?
No!
Boldly Go where no Semantics has Gone Before

Abstract away from Unknowables during Semantic Evaluation
▶ Don’t know which branch is taken — Generate all—cumulative semantics—with appropriate path condition
▶ Don’t know values of parameters, attributes, initial state — Use symbolic values: inspired by symbolic execution
▶ Don’t know identity of this, destiny — Render semantic evaluation parametric in current object, future
No!
Boldly Go where no Semantics has Gone Before

\[ v = f{.get}; \text{ if } (v == 0) \text{ then } o.m() \text{ else } \text{ await } f' ? \]

Abstract away from Unknowables during Semantic Evaluation
No!
Boldly Go where no Semantics has Gone Before

$v = f.g$et; if $(v == 0)$ then $o.m()$ else await $f'$?

Abstract away from Unknowables during Semantic Evaluation

▶ Don’t know which branch is taken — Generate all—cumulative semantics—with appropriate path condition
No!
Boldly Go where no Semantics has Gone Before

\[ v = f.g\text{et}; \ \text{if} \ (v == 0) \ \text{then} \ o.m() \ \text{else} \ \text{await} \ f' \]

Abstract away from Unknowables during Semantic Evaluation

▶ Don’t know which branch is taken —
Generate all—cumulative semantics—with appropriate path condition
▶ Don’t know values of parameters, attributes, initial state —
Use symbolic values: inspired by symbolic execution
No!
Boldly Go where no Semantics has Gone Before

\[ v = f\.get; \text{ if } (v == 0) \text{ then } o.m() \text{ else } \text{await } f' \]

Abstract away from Unknowables during Semantic Evaluation

- Don’t know which branch is taken —
  Generate all—cumulative semantics—with appropriate path condition

- Don’t know values of parameters, attributes, initial state —
  Use symbolic values: inspired by symbolic execution

- Don’t know identity of this, destiny —
  Render semantic evaluation parametric in current object, future
Definition (Semantic Evaluation)

For each object $O$, future $F$, ABS statement $s$, symbolic state $\sigma$

$$\text{val}_{\sigma}^{O,F}(s)$$

is a semantic evaluation function yielding a set of symbolic traces starting in $\sigma$. 

Symbolic, Conditioned Traces

Definition (Semantic Evaluation)
For each object $O$, future $F$, ABS statement $s$, symbolic state $\sigma$

$$\text{val}_{\sigma}^{O,F}(s)$$

is a semantic evaluation function yielding a set of symbolic traces starting in $\sigma$.

Definition (Path Condition, Symbolic Trace)
A path condition $pc$ is a set of quantifier-free formulas over $\text{Exp}(\mathcal{L})$, where $\mathcal{L}$ are memory locations (variables) and $\text{Exp}(\mathcal{L})$ ABS expressions over $\mathcal{L}$.

A (conditioned) symbolic trace has the form $pc \triangleright \tau$, where $\tau$ is a finite or infinite sequence of symbolic states.
Symbolic, Conditioned Traces

Definition (Semantic Evaluation)
For each object $O$, future $F$, ABS statement $s$, symbolic state $\sigma$

$$\text{val}_{\sigma}^{O,F}(s)$$

is a semantic evaluation function yielding a set of symbolic traces starting in $\sigma$.

Definition (Path Condition, Symbolic Trace)
A path condition $pc$ is a set of quantifier-free formulas over $\text{Exp}(\mathcal{L})$, where $\mathcal{L}$ are memory locations (variables) and $\text{Exp}(\mathcal{L})$ ABS expressions over $\mathcal{L}$.

A (conditioned) symbolic trace has the form $pc \triangleright \tau$, where $\tau$ is a finite or infinite sequence of symbolic states.

Definition (Symbolic State)
A symbolic state is a function $\sigma : \mathcal{L} \rightarrow \text{Exp}(\mathcal{L})$. 
Symbolic Traces
Conventions, Notation, Example

- Assume $\sigma(\ell) \in \text{Exp}(\mathcal{L})$ is always fully evaluated
  $\Rightarrow$ if $\sigma(\ell)$ contains no symbol from $\mathcal{L}$ then $\sigma(\ell) \in D$
- Likewise, symbol-free path condition is either “true” or “false”
- Identify true $\triangleright \tau$ with $\tau$
- Extend trace with single successor state: $\tau \fold \sigma$
- Lifting states to singleton traces: $\langle \sigma \rangle$
- Denote $\sigma(\ell) = e$ with $\ell \mapsto e$

\[
\{(v_y \neq 0)\} \triangleright \langle [O.i \mapsto v_0, y \mapsto v_y, l \mapsto v_1] \rangle \fold [O.i \mapsto 42, y \mapsto v_0 + v_y, l \mapsto v_1]
\]

- Empty trace denoted $\varepsilon$
- Concatenation of traces $\tau \cdot \omega$ (only defined if $\tau$ finite)
And Now ... Let's Go!
Local Semantics of ABS
Block Scopes

Scopes
ABS has block statements that define variable scopes: \( \{s\} \), where \( s \) a statement

Evaluation of block scopes without local variable declarations

\[
\text{val}_\sigma^O \cdot F (\{s\}) =
\]
Scopes

ABS has block statements that define variable scopes: \{s\}, where s a statement

Evaluation of block scopes without local variable declarations

\[
\text{val}_{\sigma}^{O,F}(\{s\}) = \text{val}_{\sigma}^{O,F}(s)
\]
Local Semantics of ABS
Block Scopes

Scopes

ABS has block statements that define variable scopes: \{s\}, where s a statement

Evaluation of block scopes without local variable declarations

\[ \text{val}^{O,F}_\sigma (\{s\}) = \text{val}^{O,F}_\sigma (s) \]

Wlog, local variable declarations appear only at the beginning of block scopes

\[ \text{val}^{O,F}_\sigma (\{ T \ell = e; bs\}) = \]

Local Semantics of ABS
Block Scopes

Scopes
ABS has block statements that define variable scopes: \{s\}, where s a statement

Evaluation of block scopes without local variable declarations

\[ \text{val}_{\sigma}^{O,F}(\{s\}) = \text{val}_{\sigma}^{O,F}(s) \]

Wlog, local variable declarations appear only at the beginning of block scopes

\[ \text{val}_{\sigma}^{O,F}(\{ T \; \ell = e; \; bs \}) = \]

\[ \text{pc} \triangleright \omega \in \text{val}_{\sigma}^{O,F}(\{bs[\ell' / \ell]\}), \text{isFresh}(\ell') \]

28 May 2018 | TU Darmstadt, Software Engineering | Reiner Hähnle | 15
Local Semantics of ABS
Block Scopes

Scopes

ABS has block statements that define variable scopes: \{s\}, where s a statement

Evaluation of block scopes without local variable declarations

\[
\text{val}^{O,F}_\sigma (\{s\}) = \text{val}^{O,F}_\sigma (s)
\]

Wlog, local variable declarations appear only at the beginning of block scopes

\[
\text{val}^{O,F}_\sigma (\{ T \ell = e; bs \}) = \\
\sigma' = \sigma [\ell' \mapsto \text{val}^{O,F}_\sigma (e)], \\
pc \triangleright \omega \in \text{val}^{O,F}_{\sigma'} (\{bs[\ell' / \ell]\}), \text{isFresh}(\ell')
\]
ABS has block statements that define variable scopes: \{s\}, where s a statement

Evaluation of block scopes without local variable declarations

\[ \text{val}^{O,F}_{\sigma}(\{s\}) = \text{val}^{O,F}_{\sigma}(s) \]

Wlog, local variable declarations appear only at the beginning of block scopes

\[ \text{val}^{O,F}_{\sigma}(\{T\ell = e; bs\}) = \{pc \triangleright \langle \sigma \rangle \cdot \omega \mid \sigma' = \sigma[\ell' \mapsto \text{val}^{O,F}_{\sigma}(e)], \text{pc} \triangleright \omega \in \text{val}^{O,F}_{\sigma}(\{bs[\ell' / \ell]\}), \text{isFresh}(\ell') \} \]
Local Semantics of ABS
Skip and Local Variable Assignment

\[ \text{val}^{O,F}_{\sigma}\text{(skip)} = \{ \emptyset \triangleright \langle \sigma \rangle \} \]

\[ \text{val}^{O,F}_{\sigma}\text{(l = e)} = \{ \emptyset \triangleright \langle \sigma \rangle \bowtie \sigma[\ell \mapsto \text{val}^{O,F}_{\sigma}\text{(e)}] \} \]
Local Semantics of ABS
Skip and Local Variable Assignment

\[
\text{val}_{\sigma}^{O,F}(\text{skip}) = \{ \emptyset \triangleright \langle \sigma \rangle \}
\]

\[
\text{val}_{\sigma}^{O,F}(\ell = e) = \{ \emptyset \triangleright \langle \sigma \rangle \bowtie \sigma[\ell \mapsto \text{val}_{\sigma}^{O,F}(e)] \}
\]
Local Semantics of ABS
Skip and Local Variable Assignment

val_{O,F}^{\sigma}(\text{skip}) = \{\emptyset \triangleright \langle \sigma \rangle\}

val_{O,F}^{\sigma}(\ell = e) = \{\emptyset \triangleright \langle \sigma \rangle \bowtie \sigma[\ell \mapsto \text{val}_{O,F}^{\sigma}(e)]\}

Local Semantics of ABS
Skip and Local Variable Assignment

\[ \text{val} \ O, \ F \sigma (\text{skip}) = \{ \emptyset \ \triangleleft \langle \sigma \rangle \} \]

\[ \text{val} \ O, \ F \sigma (\ell = e) = \{ \emptyset \ \triangleleft \langle \sigma \rangle \ \triangleright \sigma [\ell \mapsto \text{val} \ O, \ F \sigma (e)] \} \]
Recall the ABS Communication Structure

- **Guest this**
- **order(d) on w**
- **prepare(d) on c**

begin execution of eat

**Fut**: Meal \( fm = w!order(d) \)

begin execution of order(d)

**Fut**: Meal \( fc = c!prepare(d) \)

begin execution of prepare(d)

**await**: \( fc? \)

**get**: \( fc \)

**termination**

**delay**

**await**: \( fm? \)

**get**: \( fm \)

**termination**

28 May 2018 | TU Darmstadt, Software Engineering | Reiner Hähnle | 17
Incorporate Communication Events into Traces

Lifting Event Markers to Traces

Let $ev(\overline{v})$ be an event marker with arguments $\overline{v}$

How to associate $ev(\overline{v})$ with a state $\sigma$ inside a trace $\tau$?

$$ev_{\sigma}(\overline{v}) = \langle \sigma \rangle \circlearrowleft ev(\overline{v}) \circlearrowleft \sigma$$

- Event trace $ev_{\sigma}(\overline{v})$ is a trace of length 3
- Advantage: Traces begin and end always with states
Incorporate Communication Events into Traces

Lifting Event Markers to Traces

Let $ev(\overline{v})$ be an event marker with arguments $\overline{v}$
How to associate $ev(\overline{v})$ with a state $\sigma$ inside a trace $\tau$?

$$ev_\sigma(\overline{v}) = \langle \sigma \rangle \circlearrowleft ev(\overline{v}) \circlearrowright \sigma$$

Event trace $ev_\sigma(\overline{v})$ is a trace of length 3

Advantage: Traces begin and end always with states

Event markers will be used to ensure well-formedness of traces: objects must be created before they can be accessed, etc.
class \( C(\overline{T} \, \overline{a}) \) implements I { ... }

\( \ell = \textbf{new} \ C(\overline{e}) \); // create new object of class \( C \)
// with class attribute arguments \( \overline{e} \) and assign to \( \ell \)
Local Semantics of ABS
Object Creation

class \( C(T \overline{a}) \) implements I { ... }
\( \ell = \text{new} \ C(\overline{e}) \);  // create new object of class C
// with class attribute arguments \( \overline{e} \) and assign to \( \ell \)

Object Initialization

► Wlog no initialization block

\[
\text{val}_{\sigma}^{O,F}(\ell = \text{new} \ C(\overline{e})) =
\]
Local Semantics of ABS
Object Creation

class C(T a) implements I { ... }
ℓ = new C(ė); // create new object of class C
// with class attribute arguments ė and assign to ℓ

Object Initialization

▶ Wlog no initialization block
▶ For fresh object o the initial state C.ε(o, V):
  sets each attribute to fresh symbol, class attributes to constructor values V

{ pc ⊥ newEvσ(this, newObject, attributeValues) | isFresh(o), class(o) = C, σ' = C.ε(o, V) \circ σ, V = val^{O,F}_σ(ė) }
Local Semantics of ABS
Object Creation

```java
class C(T a) implements I { ... }
ℓ = new C(e); // create new object of class C
// with class attribute arguments e and assign to ℓ
```

**Object Initialization**

- Wlog no initialization block
- For fresh object o the initial state \( C.ɛ(o, \overline{V}) \): sets each attribute to fresh symbol, class attributes to constructor values \( \overline{V} \)

\[
\text{val}_{σ}^{O,F}(ℓ = \text{new } C(\overline{e})) = \begin{cases} 
\text{isFresh}(o), & \text{class}(o) = C, \ σ' = C.ɛ(o, \overline{V}) \circ σ, \\
pC \triangleright τ \in \text{val}_{σ}^{O,F}(ℓ = o), \ \overline{V} = \text{val}_{σ}^{O,F}(\overline{e})
\end{cases}
\]
Local Semantics of ABS
Object Creation

class \( C(\overline{T}, \overline{a}) \) implements I { ... }
\[ \ell = \texttt{new} \ C(\overline{e}); \quad // \text{create new object of class } C \]
\[ // \text{with class attribute arguments } \overline{e} \text{ and assign to } \ell \]

Object Initialization

- Wlog no initialization block
- For fresh object \( o \) the initial state \( C.\epsilon(o, \overline{v}) \):
  sets each attribute to fresh symbol, class attributes to constructor values \( \overline{v} \)
- Event marker \( \texttt{newEv}_\sigma(\texttt{this}, \texttt{newObject}, \texttt{attributeValues}) \)

\[
\text{val}^O,F_{\sigma}(\ell = \texttt{new} \ C(\overline{e})) = \{ pc \triangleright \texttt{newEv}_\sigma(O, o, \overline{v}) \cdot \tau \mid
\]
\[ isFresh(o), \ class(o) = C, \ \sigma' = C.\epsilon(o, \overline{v}) \circ \sigma, \]
\[ pc \triangleright \tau \in \text{val}^O,F_{\sigma}(\ell = o), \ \overline{v} = \text{val}^O,F_{\sigma}(\overline{e}) \} \]
Local Semantics of ABS
Asynchronous Method Call

\[ \ell = e'!m(\overline{e}); \quad \text{// asynchronous call of } m \text{ on } e' \text{ with arguments } \overline{e} \]
\[ \text{// assign result to } \ell \]
Local Semantics of ABS
Asynchronous Method Call

ℓ = e′!m(ē); // asynchronous call of m on e' with arguments ē
// assign result to ℓ

Event Marker

Invocation event \( \text{invEv}_\sigma(\text{caller}, \text{callee}, \text{future}, \text{method}, \text{args}) \)
Local Semantics of ABS
Asynchronous Method Call

\[ \ell = e'!m(\overline{e}); \quad // \text{asynchronous call of } m \text{ on } e' \text{ with arguments } \overline{e} \]
\[ // \text{assign result to } \ell \]

Event Marker

Invocation event \( \text{invEv}_\sigma(caller, callee, future, method, args) \)

\[ \text{val}^{O,F}_\sigma(\ell = e'!m(\overline{e})) = \]

\[ \text{isFresh}(f), \text{method}(f) = m, \text{pc} \triangleq \tau \in \text{val}^{O,F}_\sigma(\ell = f) \]
Local Semantics of ABS
Asynchronous Method Call

\[ \ell = e'!m(\bar{e}); \quad // \text{asynchronous call of } m \text{ on } e' \text{ with arguments } \bar{e} \\
// \text{assign result to } \ell \]

Event Marker

Invocation event \( \text{invEv}_\sigma(caller, callee, future, method, args) \)

\[
\text{val}^{O,F}_\sigma(\ell = e'!m(\bar{e})) = \\
isFresh(f), \text{ method}(f) = m, pc \triangleright \tau \in \text{val}^{O,F}_\sigma(\ell = f)
\]
Local Semantics of ABS
Asynchronous Method Call

\[ \ell = e'!m(\bar{e}); \quad // \text{asynchronous call of } m \text{ on } e' \text{ with arguments } \bar{e} \]

// assign result to \( \ell \)

Event Marker

Invocation event \( \text{invEv}_\sigma(caller, callee, future, method, args) \)

\[ \text{val}^{O,F}_\sigma(\ell = e'!m(\bar{e})) = \{ pc \triangleright \text{invEv}_\sigma(O, \text{val}^{O,F}_\sigma(e'), f, m, \text{val}^{O,F}_\sigma(\bar{e})) \; \quad \ast\ast \; \tau \mid \]

isFresh(f), method(f) = m, pc \triangleright \tau \in \text{val}^{O,F}_\sigma(\ell = f) \} \]
Local Semantics of ABS
Asynchronous Method Call

\[ \ell = e'!m(\overline{e}); \quad // \text{asynchronous call of } m \text{ on } e' \text{ with arguments } \overline{e} \]
\[ // \text{assign result to } \ell \]

Event Marker
Invocation event \( invEv_\sigma(caller, callee, future, method, args) \)

\[
val^{O,F}_\sigma(\ell = e'!m(\overline{e})) = \{pc \triangleright \ invEv_\sigma(O, val^{O,F}_\sigma(e'), f, m, val^{O,F}_\sigma(\overline{e})) \quad ** \quad \tau \mid \\
\quad isFresh(f), \ method(f) = m, \ pc \triangleright \tau \in val^{O,F}_\sigma(\ell = f) \}
\]

What is “**”? 
The “Chop” Constructor for Traces

Semantics of the Sequencing Statement in Terms of Traces

\[ r; s \]
The “Chop” Constructor for Traces

Semantics of the Sequencing Statement in Terms of Traces

\[ r; s \]

\[
\begin{array}{c}
\sigma_0 \cdots \sigma_n \\
\underbrace{\cdots} \\
r
\end{array}
\quad
\begin{array}{c}
\sigma_n \sigma_{n+1} \cdots \\
\underbrace{\cdots} \\
s
\end{array}
\]

If \( \tau \) is infinite, returns \( \tau \), otherwise defined as above.

Event lifting:
\[ \text{ev} (\sigma) = \langle \sigma \rangle \rightarrow \text{ev} (v) \rightarrow \sigma : \text{events are “choppable”} \]
The “Chop” Constructor for Traces

Semantics of the Sequencing Statement in Terms of Traces

\[ r; s \]

\[
\sigma_0 \cdots \sigma_n \quad ** \quad \sigma_n \sigma_{n+1} \cdots
\]

\( r \)

\( s \)

▶ \( \tau ** \tau' \) is “chop” on traces: cut out one redundant state
The “Chop” Constructor for Traces

Semantics of the Sequencing Statement in Terms of Traces

\[ r; s \]

\[
\underbrace{\sigma_0 \cdots \sigma_n}_{r} \quad \parallel \quad \underbrace{\sigma_n \sigma_{n+1} \cdots}_{s}
\]

▶ \( \tau \兴建 \tau' \) is “chop” on traces: cut out one redundant state
The “Chop” Constructor for Traces

Semantics of the Sequencing Statement in Terms of Traces

$r; s$

\[
\sigma_0 \cdots \sigma_n \sigma_{n+1} \cdots
\]

\[
\sigma_0 \cdots \sigma_n \quad \text{**} \quad \sigma_n \sigma_{n+1} \cdots
\]

\[
r \quad \text{||} \quad s
\]

- $\tau \text{**}\tau'$ is “chop” on traces: cut out one redundant state
- If $\tau$ is infinite, returns $\tau$, otherwise defined as above
The “Chop” Constructor for Traces

Semantics of the Sequencing Statement in Terms of Traces

\[ r; s \]

\[ \sigma_0 \cdots \sigma_n \sigma_{n+1} \cdots \]

\[ r \]

\[ \sigma_0 \cdots \sigma_n \]

\[ ** \]

\[ \sigma_n \sigma_{n+1} \cdots \]

\[ s \]

- \( \tau \ ** \ \tau' \) is “chop” on traces: cut out one redundant state
- If \( \tau \) is infinite, returns \( \tau \), otherwise defined as above
- Event lifting \( ev_{\sigma}(\vec{v}) = \langle \sigma \rangle \bowtie ev(\vec{v}) \bowtie \sigma \): events are “choppable”
Local Semantics of ABS
Method Execution

Event Marker

Invocation reaction event $\text{invREv}_\sigma(\text{caller}, \text{callee}, \text{future}, \text{method}, \text{args})$

$$\text{val}^{O,F}_{\sigma} (C.m) =$$
Local Semantics of ABS
Method Execution

Event Marker

Invocation reaction event $\text{invREv}_\sigma(caller, callee, future, method, args)$

\[
\text{val}^{O,F}_\sigma(C.m) = \\
\text{lookup}(m, C) = T m(T \overline{\ell'})\{s\},
\]
Local Semantics of ABS
Method Execution

Event Marker

Invocation reaction event \( \text{invREv}_\sigma(\text{caller}, \text{callee}, \text{future}, \text{method}, \text{args}) \)

\[
\text{val}_\sigma^{O,F}(C.m) = \begin{align*}
\text{pc} & \triangleright \omega \in \text{val}_\sigma^{O,F}(\{ \overline{T} \overline{\ell}' = \overline{v_0} ; s \}), \ \text{isFresh}(O', \overline{v_0}) \\
\text{lookup}(m, C) & = T \ m(\overline{T} \overline{\ell}') \{ s \},
\end{align*}
\]

- Unknown parameter values initialized with fresh symbolic constants \( \overline{v_0} \)
- Call parameters inside scope: no name clash
Local Semantics of ABS
Method Execution

Event Marker

Invocation reaction event \( \text{invREv}_\sigma(caller, callee, future, method, args) \)

\[
\text{val}^{O,F}_\sigma(C.m) = \{ pc \triangleright \text{invREv}_\sigma(O', O, F, m, \overline{v_0}) \star \omega \mid pc \triangleright \omega \in \text{val}^{O,F}_\sigma(\{ \overline{T \overline{l'}} = \overline{v_0}; s \}), \text{isFresh}(O', \overline{v_0}) \}
\]

\[
\text{lookup}(m, C) = T \; m(\overline{T \overline{l'}})\{s\}, \}
\]

- Unknown parameter values initialized with fresh symbolic constants \( \overline{v_0} \)
- Call parameters inside scope: no name clash
- Unknown caller initialized with fresh parameter \( O' \)
Local Semantics of ABS
Method Execution

Event Marker

Invocation reaction event \( \text{invREv}_\sigma(caller, callee, future, method, args) \)

\[
\begin{align*}
\text{val}^{O,F}_\sigma(C.m) &= \{ pc > \text{invREv}_\sigma(O', O, F, m, \overline{v}_0) \, ** \, \omega \mid \\
& \quad pc > \omega \in \text{val}^{O,F}_\sigma(\{ T \, \overline{l}' = \overline{v}_0; s \}), \text{isFresh}(O', \overline{v}_0) \\
& \quad \text{lookup}(m, C) = T \, m(\overline{T \, \overline{l}'})\{s\}, \}
\end{align*}
\]

- Unknown parameter values initialized with fresh symbolic constants \( \overline{v}_0 \)
- Call parameters inside scope: no name clash
- Unknown caller initialized with fresh parameter \( O' \)

Conditional, method return, synchronous calls: straightforward
Local Semantics of ABS
Method Execution

Event Marker

Invocation reaction event $\text{invREv}_\sigma(caller, callee, future, method, args)$

$$
\text{val}_{O,F}^{{O}',m} = \{ pc \triangleright \text{invREv}_\sigma(O', O, F, m, v_0) \} \quad \text{where}
$$
$$
\begin{align*}
pc & \triangleright \omega \in \text{val}_{O,F}^{{O}',m}(\{ T \ell = v_0; s \}), \quad \text{isFresh}(O', v_0) \\
\text{lookup}(m, C) & = T m(T \ell')\{ s \}
\end{align*}
$$

▶ Unknown parameter values initialized with fresh symbolic constants $v_0$
▶ Call parameters inside scope: no name clash
▶ Unknown caller initialized with fresh parameter $O$

Conditional, method return, synchronous calls: straightforward
Local Semantics of ABS Method Execution

Event Marker

Invocation reaction event

\[ \sigma(caller, callee, future, method, args) \]

\[ \text{val}_{\sigma}(C, m) = \{ \text{pc} \doteq \text{invREv}_\sigma(O', O, F, m, v_0) \}^{\omega} \]

\[ \text{lookup}(m, C) = T_m(T_{\ell'} = v_0; s) \]

Unknown parameter values initialized with fresh symbolic constants \( v_0 \)

Call parameters inside scope: no name clash

Unknown caller initialized with fresh parameter \( O' \)

Conditional, method return, synchronous calls: straightforward
Suspension and Resumption

Problems with Release of Control and Interleaving

1. Impossible to know the computation state after resumption
2. When composing behavior, we need to know interleaving points

\[ \text{val}_{\sigma}^{O,F}(\text{suspend}) = \]
Suspension and Resumption

Problems with Release of Control and Interleaving

1. Impossible to know the computation state after resumption
2. When composing behavior, we need to know interleaving points

Make use of release events and continuations

\[ \text{val}^{O,F}_\sigma (\text{suspend}) = \{ \emptyset \triangleright \text{relEv}_\sigma (O) \cdot \text{relCont}(O, F, \text{skip}) \} \]

- \text{relCont} is not state/event, but continuation marker to store future behavior
Suspension and Resumption

Problems with Release of Control and Interleaving

1. Impossible to know the computation state after resumption
2. When composing behavior, we need to know interleaving points

Make use of release events and continuations

\[
\text{val}_{\sigma}^{O,F}(\text{suspend}) = \{ \emptyset \triangleright relEv_\sigma(O) \cdot starve(O) \} \cup \\
\{ \emptyset \triangleright relEv_\sigma(O) \cdot relCont(O, F, \text{skip}) \}
\]

- \( relCont \) is not state/event, but continuation marker to store future behavior
- Process might never be re-scheduled: causes different global behavior
- Starvation marker (only at end of a trace) signifies this
- Continuation and starvation markers are not part of trace
In a local semantics, sequential composition becomes tricky

\[
\text{val}_{\sigma}^{O,F} (r; s) = \{
\begin{array}{l}
\text{pc}_r \trianglerighteq \tau_r \\
\text{pc}_s \trianglerighteq \omega_s
\end{array}
\}
\}
\cup \{
\text{pc}_r \trianglerighteq \tau_r \cdot \text{relCont} (O, F, r'; s) \\
\text{pc}_r \trianglerighteq \tau_r \cdot \text{relCont} (O, F, r') \in \text{val}_{\sigma}^{O,F} (r)
\}
\]

- Non-termination, starving handled in definition of \(\ast\): throw away \(\omega_s\)
- \(\tau_r\) must end with continuation marker: compose with \(s\)
In a local semantics, sequential composition becomes tricky.

\[ \text{val}_{O,F}^\sigma (r; s) = \]
\[ \{ (pc_r \triangleright \tau_r) \ast\ast (pc_s \triangleright \omega_s) \mid pc_r \triangleright \tau_r \in \text{val}_{O,F}^\sigma (r), pc_s \triangleright \omega_s \in \text{val}_{O,F}^\sigma (s), \]
\[ \text{where } \sigma' = \text{last}(\tau_r) \text{ if } \tau_r \text{ is finite} \}

\]
In a local semantics, sequential composition becomes tricky

- Execution of \( r \) might diverge or starve

\[
\text{val}_{\sigma}^{O,F}(r; s) = \left\{ (pc_r \triangleright \tau_r) \quad \star\star \quad (pc_s \triangleright \omega_s) \mid pc_r \triangleright \tau_r \in \text{val}_{\sigma}^{O,F}(r), \; pc_s \triangleright \omega_s \in \text{val}_{\sigma'}^{O,F}(s), \right. \\
\left. \quad \text{where } \sigma' = \text{last}(\tau_r) \text{ if } \tau_r \text{ is finite} \right\}
\]
Semantics of Sequential Composition — Continuation Passing Style

In a local semantics, sequential composition becomes tricky

- Execution of $r$ might diverge or starve

$$\text{val}_{\sigma}^{O,F}(r; s) = \{ (pc_r \triangleright \tau_r) \star\star (pc_s \triangleright \omega_s) \mid pc_r \triangleright \tau_r \in \text{val}_{\sigma}^{O,F}(r), \ pc_s \triangleright \omega_s \in \text{val}_{\sigma'}^{O,F}(s),$$
$$\text{where } \sigma' = \text{last}(\tau_r) \text{ if } \tau_r \text{ is finite, arbitrary otherwise} \}$$

- Non-termination, starving handled in definition of $\star\star$: throw away $\omega_s$
Semantics of Sequential Composition — Continuation Passing Style

In a local semantics, sequential composition becomes tricky

- Execution of $r$ might diverge or starve
- $r$ may contain release points

\[
\text{val}_{\sigma}^{O,F}(r; s) = \{
(p_{cr} \triangleright \tau_r) \ast\ast (p_{cs} \triangleright \omega_s) \mid p_{cr} \triangleright \tau_r \in \text{val}_{\sigma}^{O,F}(r),
\ p_{cs} \triangleright \omega_s \in \text{val}_{\sigma'}^{O,F}(s),
\text{where } \sigma' = \text{last}(\tau_r) \text{ if } \tau_r \text{ is finite, arbitrary otherwise}\}
\]

- Non-termination, starving handled in definition of $\ast\ast$: throw away $\omega_s$
In a local semantics, sequential composition becomes tricky

- Execution of $r$ might diverge or starve
- $r$ may contain release points

$$\text{val}^{O,F}_{\sigma}(r; s) = $$

\[ \{ (pc_r \triangleright \tau_r) \ast\ast (pc_s \triangleright \omega_s) \mid pc_r \triangleright \tau_r \in \text{val}^{O,F}_{\sigma}(r), \ pc_s \triangleright \omega_s \in \text{val}^{O,F}_{\sigma'}(s), \text{ where } \sigma' = \text{last}(\tau_r) \text{ if } \tau_r \text{ is finite, arbitrary otherwise}\} \cup \]

\[ \{ pc_r \triangleright \tau_r \cdot \text{relCont}(O, F, r'; s) \mid pc_r \triangleright \tau_r \cdot \text{relCont}(O, F, r') \in \text{val}^{O,F}_{\sigma}(r) \} \]

- Non-termination, starving handled in definition of $\ast\ast$: throw away $\omega_s$
- $\tau_r$ must end with continuation marker: compose with $s$
Semantic Evaluation of Sequential Composition

Example

\[ \text{val}^{O,F}_{\sigma} (\text{suspend}; s) = \]
Semantic Evaluation of Sequential Composition

Example

\[ \text{val}_{\sigma}^{O,F}(\text{suspend}; s) = \]

\[ \text{val}_{\sigma}^{O,F}(\text{suspend}) = \{ \emptyset \triangleright \text{relEv}_{\sigma}(O) \cdot \text{starve}(O) \} \cup \]
\[ \{ \emptyset \triangleright \text{relEv}_{\sigma}(O) \cdot \text{relCont}(O, F, \text{skip}) \} \]
Semantic Evaluation of Sequential Composition

Example

\[ \text{val}_{\sigma}^{O,F} (\text{suspend}; s) = \{ \emptyset \triangleright \text{relEv}_{\sigma}(O) \cdot \text{starve}(O) \} \cup \]

\[ \text{val}_{\sigma}^{O,F} (\text{suspend}) = \{ \emptyset \triangleright \text{relEv}_{\sigma}(O) \cdot \text{starve}(O) \} \cup \]

\[ \{ \emptyset \triangleright \text{relEv}_{\sigma}(O) \cdot \text{relCont}(O, F, \text{skip}) \} \]
Semantic Evaluation of Sequential Composition

Example

\[
\text{val}_{\sigma}^{O,F} (\text{suspend}; s) = \{ \emptyset \triangleright \text{relEv}_{\sigma}(O) \cdot \text{starve}(O) \} \cup \\
\{ \emptyset \triangleright \text{relEv}_{\sigma}(O) \cdot \text{relCont}(O, F, \text{skip}; s) \}
\]

\[
\text{val}_{\sigma}^{O,F} (\text{suspend}) = \{ \emptyset \triangleright \text{relEv}_{\sigma}(O) \cdot \text{starve}(O) \} \cup \\
\{ \emptyset \triangleright \text{relEv}_{\sigma}(O) \cdot \text{relCont}(O, F, \text{skip}) \}
\]
Computation of Local Traces
Example

```cpp
Int n() {
    Int y = 10;
    Fut<Int> l = 0;
    l = this!m();
    if (y == 0) then y = this.m() else await l? fi;
    y = this.i + y;
    return y;
}
```

\[\text{val}^{O,F}_{C.\epsilon(O)}(C.n) = \{\}
\]
Int n() {
    Int y = 10;
    Fut<Int> l = 0;
    l = this!m();
    if (y == 0) then y = this.m() else await ? fi;
    y = this.i + y;
    return y;
}

\[
\text{val}^{O,F}_{C,e(O)}(C.n) = \{ \\
\{(10 = 0)\} \triangleright \langle [O.i \mapsto v_i] \bowtie \text{invREv}(O', O, F, n, _) \bowtie \cdots \text{ infeasible path condition} \\
\}
\]
Computation of Local Traces
Example

```clojure
Int n() {
    Int y = 10;
    Fut<Int> l = 0;
    l = this!m();
    if (y == 0) then y = this.m() else await l? fi;
    y = this.i + y;
    return y;
}
```

\[
\text{val}^{O,F}_{C.\epsilon(O)}(C.n) = \{
\{(10 \neq 0)\} \triangleright \langle [O.i \mapsto v_i]\rangle \bowtie \text{invREv}(O', O, F, n, \_)) \bowtie \cdots \bowtie \text{futEv}(O, F, v_i + 10) \bowtie [O.i \mapsto v_i, \ y' \mapsto v_i + 10, \ l' \mapsto f_0]\}
\]

▶ Future \(f_0\) in \(l\) already resolved at `await`: \(n\) runs to completion
Computation of Local Traces

Example

```c
Int n() {
    Int y = 10;
    Fut<Int> l = 0;
    l = this!m();
    if (y == 0) then y = this.m() else await l? fi;
    y = this.i + y;
    return y;
}
```

val\textsubscript{C.\epsilon(O)}(C.n) = \{
{(10 \neq 0)} \triangleright \langle [O.i \mapsto v_i] \rangle \bowtie \textit{invREV}(O', O, F, n, _) \bowtie \cdots \textit{starve}(O)
\}

▶ n starves at \textit{await}
Computation of Local Traces

Example

```java
Int n() {
    Int y = 10;
    Fut<Int> l = 0;
    l = this!m();
    if (y == 0) then y = this.m() else await l? fi;
    y = this.i + y;
    return y;
}
```

\[
\text{val}^{O,F}_{C\in\{O\}}(C.n) = \{
    \{(10 \neq 0)\} \triangleright \langle [O.i \mapsto v_i] \bowtie \cdots \bowtie \text{relEv}(O, f_0) \bowtie [O.i \mapsto v_i, y' \mapsto 10, l' \mapsto f_0] \\
    \cdot \text{relCont}(O, F, \text{await} l'?; y' = \text{this}.i + y'; \text{return} y'; )
\}
\]

- Release control at `await`: put remaining code in continuation
Goal

Given a main block \( \{ \overline{T} \overline{\ell} = \overline{v}; s \} \) and ABS program \( P \) produce all valid, concrete system traces
From Local to Global Traces

Goal

Given a main block \( \{ T \ell = \nu; s \} \) and ABS program \( P \)
produce all valid, concrete system traces

1. Compute local traces of main block: \( M = \text{val}^{\text{Main}, f_0}_\epsilon (\{ T \ell = \nu; s \}) \)
**Goal**

Given a main block \( \{ \overline{T} \ell = \overline{v}; s \} \) and ABS program \( P \) produce all valid, concrete system traces

1. Compute local traces of main block: \( \mathcal{M} = \text{val}^{\text{Main}, f_0}(\{ \overline{T} \ell = \overline{v}; s \}) \)

Result: initial concrete, non-empty traces with path condition true or false
From Local to Global Traces

Goal

Given a main block \( \{ T \ell = \bar{v}; s \} \) and ABS program \( P \) produce all valid, concrete system traces

1. Compute local traces of main block: \( \mathcal{M} = \text{val}^{\text{Main}, f_0}(\{ T \ell = \bar{v}; s \}) \)

Result: initial concrete, non-empty traces with path condition true or false

2. Compute local traces of each method for all objects and futures:

\[
\mathcal{G} = \{ \text{val}^{O,F}_{C,O}(C.m) \mid \text{class}(O) = C, \ m \in \text{mtd}(C), \ O \in \mathcal{O}, \ F \in \mathcal{F}, \ C \in \mathcal{P} \}
\]

3. Pick an initial concrete trace with path condition true from \( \mathcal{M} \) and extend it with suitable instances from \( \mathcal{G} \), repeat
Definition (Global Trace Composition Rule)

Let $sh$ be a finite concrete trace and $q$ a pool (queue) of sets of symbolic traces. A global trace composition rule has the form

$$\begin{align*}
\text{Conditions on } sh, q \quad \quad \\
\frac{}{sh, q \rightarrow sh', q'}
\end{align*}$$

- Any exhaustive application of global trace composition rules yields one valid, global system trace, possibly infinite
- Initial configuration is: $\varepsilon, \{M\} \cup G$
How to Preempt Local Execution?

ABS has no preemption . . .
Interleave execution on different processors with *interleaving events*
External Interleaving

How to Preempt Local Execution?

ABS has no preemption . . .
Interleave execution on different processors with **interleaving events**

Execute arbitrary finite local trace, then interleave other process
External Interleaving

How to Preempt Local Execution?

ABS has no preemption . . .
Interleave execution on different processors with interleaving events

Execute arbitrary finite local trace, then interleave other process

\[ \Omega \in q \quad object(\Omega) = O \quad pc > \tau \cdot \omega \in \Omega \quad \text{get symbolic trace on an } O \text{ from pool } q \]

\[
\begin{align*}
sh, q & \rightarrow sh & , q' \\
\end{align*}
\]
External Interleaving

How to Preempt Local Execution?

ABS has no preemption . . .
Interleave execution on different processors with *interleaving events*

Execute arbitrary finite local trace, then interleave other process

\[ \Omega \in q \quad \text{object}(\Omega) = O \quad pc \triangleright \tau \cdot \omega \in \Omega \]

\[ \text{last}(sh) = \sigma \]

get symbolic trace on an \( O \) from pool \( q \)
get final *concrete* state \( \sigma \) from \( sh \)

\[ sh, q \rightarrow sh \quad , \quad q' \]
External Interleaving

How to Preempt Local Execution?

ABS has no preemption . . .
Interleave execution on different processors with interleave events.

Execute arbitrary finite local trace, then interleave other process

\[ \Omega \in q \quad \text{object}(\Omega) = O \quad pc \triangleright \tau \cdot \omega \in \Omega \]
\[ \text{last}(sh) = \sigma \]
\[ \tau \neq \varepsilon \]

get symbolic trace on an \( O \) from pool \( q \)
get final concrete state \( \sigma \) from \( sh \)
make some finite progress

\[ sh, q \rightarrow sh, q' \]
External Interleaving

How to Preempt Local Execution?

ABS has no preemption . . .
Interleave execution on different processors with interleaving events

Execute arbitrary finite local trace, then interleave other process

\[ \Omega \in q \quad \text{object(} \Omega \text{)} = O \quad pc \triangleright \tau \cdot \omega \in \Omega \]

\[ \text{last(} sh \text{)} = \sigma \]

\[ \tau \neq \varepsilon \]

\[ \omega \notin \{\varepsilon, \text{relCont}(O,\_,\_), \text{starve}(O)\} \]

get symbolic trace on an \( O \) from pool \( q \)
get final concrete state \( \sigma \) from \( sh \)
make some finite progress
don’t finish execution on \( O \)

\[ sh, q \rightarrow sh \quad , \quad q' \]
External Interleaving

How to Preempt Local Execution?

ABS has no preemption . . .
Interleave execution on different processors with interleaving events

Execute arbitrary finite local trace, then interleave other process

\[ \Omega \in q \quad object(\Omega) = O \quad pc \triangleright \tau \cdot \omega \in \Omega \]

last(sh) = \sigma
\[ \tau \neq \varepsilon \]
\[ \omega \not\in \{ \varepsilon, relCont(O, _, _), starve(O) \} \]

\[ pc_\sigma = \text{true} \quad wf(sh \; ** \tau_\sigma) \]

get symbolic trace on an O from pool q
get final concrete state \sigma from sh
make some finite progress
don’t finish execution on O
\sigma-instance of \( pc \triangleright \tau \) feasible, well-formed

\[ sh, q \rightarrow sh \; ** \tau_\sigma, \; q' \]
External Interleaving

How to Preempt Local Execution?

ABS has no preemption . . .
Interleave execution on different processors with **interleaving events**

**Execute arbitrary finite local trace, then interleave other process**

\[
\begin{align*}
\Omega & \in q \quad \text{object}(\Omega) = O \quad pc \triangleright \tau \cdot \omega \in \Omega \\
\text{last}(sh) & = \sigma \\
\tau & \neq \varepsilon \\
\omega & \notin \{\varepsilon, \text{relCont}(O, _, _), \text{starve}(O)\} \\
pc_{\sigma} & = \text{true} \quad \text{wf}(sh \triangleright \tau_{\sigma}) \\
q' & = q \setminus \Omega \cup \{\emptyset \triangleright \text{ilEv}_{\text{last}(\tau)}(O) \cdot \omega\}
\end{align*}
\]

get symbolic trace on an \( O \) from pool \( q \)
get final **concrete** state \( \sigma \) from \( sh \)
make some finite progress
don’t finish execution on \( O \)
\( \sigma \)-instance of \( pc \triangleright \tau \) feasible, well-formed
update pool, insert interleaving events

\[
sh, q \rightarrow sh \triangleright \tau_{\sigma} \triangleright \text{ilEv}_{\text{last}(\tau_{\sigma})}(O), q'
\]
Other Global Trace Composition Rules

1. External interleaving
2. Release
3. Continuation
4. Starvation
5. Blocking
Other Global Trace Composition Rules

1. External interleaving
2. Release
3. Continuation
4. Starvation
5. Blocking
Other Global Trace Composition Rules

1. External interleaving
2. Release
3. Continuation
4. Starvation
5. Blocking
Well-Formedness

Each global rule maintains well-formedness of trace extension: \( \text{wf}(sh \ast\ast \tau_\sigma) \)

- Needs to be checked only on finite, concrete traces
- Ensures that system event sequence is schedulable
- Predicate defined on event structure of given trace
Well-Formed Traces

Well-Formedness

Each global rule maintains well-formedness of trace extension: $wf(sh^{**} \tau_\sigma)$

- Needs to be checked only on finite, concrete traces
- Ensures that system event sequence is schedulable
- Predicate defined on event structure of given trace

Examples of Well-Formedness Conditions

- “A release event for a future $f$ cannot be preceded by a completion event for $f$”
- “An external interleaving event on $O$ must be directly followed by its corresponding interleaving reaction event”
  - This prevents local preemption
Each statement is evaluated **locally** for any object, future

- Evaluation of statement yields set of **symbolic traces**
- Evaluation is **independent** from other statements
Each statement is evaluated \textit{locally} for any object, future
  \begin{itemize}
    \item Evaluation of statement yields set of \textit{symbolic traces}
    \item Evaluation is \textit{independent} from other statements
  \end{itemize}

\textbf{Internal interleaving} realized with continuations
  \begin{itemize}
    \item Distinguish divergence, starvation, and blocking
Each statement is evaluated **locally** for any object, future
- Evaluation of statement yields set of **symbolic traces**
- Evaluation is **independent** from other statements

**Internal interleaving** realized with continuations
- Distinguish divergence, starvation, and blocking

**Global behavior** by instantiation and **external interleaving**
- Can characterize concurrency models via **well-formedness** by way of **dual events**
- Separation of concerns: computation states, event structure
Each statement is evaluated \textit{locally} for any object, future

- Evaluation of statement yields set of \textit{symbolic traces}
- Evaluation is \textit{independent} from other statements

\textbf{Internal interleaving} realized with continuations

- Distinguish divergence, starvation, and blocking

\textbf{Global behavior by instantiation and external interleaving}

- Can characterize concurrency models via \textit{well-formedness} by way of \textit{dual} events
- Separation of concerns: computation states, event structure

\textbf{One evaluation rule per statement, five global rules}
A Modular, Denotational Trace Semantics

- Each statement is evaluated \textit{locally} for any object, future
  - Evaluation of statement yields set of \textit{symbolic traces}
  - Evaluation is \textit{independent} from other statements
- Internal interleaving \textit{realized} with continuations
  - Distinguish divergence, starvation, and blocking
- Global behavior \textit{by instantiation and external interleaving}
  - Can characterize concurrency models via \textit{well-formedness} by way of \textit{dual events}
  - Separation of concerns: computation states, event structure

One evaluation rule per statement, five global rules
A Modular, Denotational Trace Semantics

- Each statement is evaluated **locally** for any object, future
  - Evaluation of statement yields set of **symbolic traces**
  - Evaluation is **independent** from other statements

- **Internal** interleaving realized with continuations
  - Distinguish divergence, starvation

- **Global** behavior by instantiation and external interleaving
  - Can characterize concurrency by way of **dual events**
  - Separation of concerns: computation states, event structure

One evaluation rule per statement, five global rules

Cheers!
Symbolic Trace Formula

An (abstract) **symbolic trace formula** evaluates to a possibly infinite set of traces
Symbolic Trace Formula

An (abstract) symbolic trace formula evaluates to a possibly infinite set of traces

“Each time a router (= this) terminates the getPk method,”

\[
\text{futEv}(\text{this}, \text{fr}, \text{getPk}, _) \]

Trace Formulas

Symbolic Trace Formula

An (abstract) symbolic trace formula evaluates to a possibly infinite set of traces.

“Each time a router terminates the `getPk` method, it must either have invoked a method to redirect a packet.”

\[
\text{invREv}(_, \text{this}, \text{fr}, \text{getPk}, (\text{pk}, _)) \quad \text{futEv}(\text{this}, \text{fr}, \text{getPk}, _) \\
1) \text{invEv}(\text{this}, \text{this}, _, \text{redirectPk}, (\text{pk}, _))
\]
An (abstract) **symbolic trace formula** evaluates to a possibly infinite set of traces.

“Each time a router terminates the `getPk` method, it must either have invoked a method to **redirect** a packet or have stored that packet in its `receivedPks` set.”

```
invREv(_,this,fr,getPk,(pk,_))

futEv(this,fr,getPk,_,)

1) invEv(this,this,_,redirectPk,(pk,_,))
2) pk in receivedPks
```
Trace Formulas

Symbolic Trace Formula

An (abstract) symbolic trace formula evaluates to a possibly infinite set of traces

“Each time a router terminates the `getPk` method, it must either have invoked a method to `redirect` a packet or have stored that packet in its `receivedPks` set”

\[
\omega_{\text{fr}, \text{pk}} \left( \text{invREv}(\_, \text{this}, \text{fr}, \text{getPk}, (\text{pk}, \_)) \iff \text{invEv}(\text{this}, \text{this}, 
\_\_, \text{redirectPk}, (\text{pk}, \_)) \iff \text{futEv}(\text{this}, \text{fr}, \text{getPk}, \_)
\right)
\]

Corresponding symbolic trace formula:
Definition (Trace Modality Formula)

1. Trace modality formulas syntactically closed under usual propositional and first-order operators.

2. If $s$ is an ABS statement and $\Psi$ a trace modality formula, then $[s]\Psi$ is a trace modality formula.

3. If $\{u\}$ is an update and $\Psi$ a trace modality formula, then $\{u\}\Psi$ is a trace modality formula.

Updates $\{\ell := \text{exp}\}$ or $\{\text{ev}(\overline{e})\}$ record state changes effected by assignments or the occurrence of communication events.
Definition (Evaluation of Trace Modality Formula)

Trace modality $\text{val}_\tau([s]\Psi)$ is true if for any $O$, $F$:

If for each $\tau' \in \text{val}_{\text{last}(\tau)}^{O,F}(s)$ such that $\tau \star\star \tau'$ is well-formed $\text{val}_{\tau \star\star \tau'}^{O,F}(\Psi)$ holds.

“Any trace of $s$ that extends $\tau$ is contained in $\Psi$”
Definition (Evaluation of Trace Modality Formula)

Trace modality $\text{val}_{\tau}([s]\Psi)$ is true if for any $O, F$:

If for each $\tau' \in \text{val}_{\text{last}(\tau)}^{O, F}(s)$ such that $\tau \ast \ast \tau'$ is well-formed $\text{val}_{\tau \ast \ast \tau'}^{O, F}(\Psi)$ holds.

“Any trace of $s$ that extends $\tau$ is contained in $\Psi$”

With a semantics for trace modality formulas, we can start to design a calculus . . .
Semantic Evaluation of Asynchronous Method Call

\[
\text{val}_{\sigma}^{O,F}(\ell = e'!m(\overline{e})) = \{pc \triangleright invEv_{\sigma}(O, \text{val}_{\sigma}^{O,F}(e'), f, m, \text{val}_{\sigma}^{O,F}(\overline{e})) \ast\ast \tau \mid \}
\]

\[
isFresh(f), \ method(f) = m, \ pc \triangleright \tau \in \text{val}_{\sigma}^{O,F}(\ell = f) \}.
\]
Semantic Evaluation of Asynchronous Method Call

$$\text{val}_{\sigma}^{O,F}(\ell = e'!m(\overline{e})) = \{ pc \triangleright inv\text{Ev}_{\sigma}(O, \text{val}_{\sigma}^{O,F}(e'), f, m, \text{val}_{\sigma}^{O,F}(\overline{e})) \triangleright\triangleright \tau \mid \text{isFresh}(f), \text{method}(f) = m, pc \triangleright \tau \in \text{val}_{\sigma}^{O,F}(\ell = f)\}$$

Sequent Rule for Asynchronous Method Call

$$\Gamma, \text{isFresh}(f) \Rightarrow U\{ \text{invEv}(\text{this}, e', f, m, \overline{e})\}\{\ell := f\}[r]\Psi$$

$$\Gamma \Rightarrow U[\ell = e'!m(\overline{e}); r]\Psi$$
Asynchronous Method Call
Semantics vs. Calculus

Semantic Evaluation of Asynchronous Method Call

\[
\text{val}_{\sigma}^{O,F}(\ell = e'!m(\overline{e})) = \{ pc \triangleright invEv_{\sigma}(O, \text{val}_{\sigma}^{O,F}(e'), f, m, \text{val}_{\sigma}^{O,F}(\overline{e})) *\ast \tau \ |
\]

\[
isFresh(f), \ method(f) = m, \ pc \triangleright \tau \in \text{val}_{\sigma}^{O,F}(\ell = f) \}
\]

Sequent Rule for Asynchronous Method Call

\[
\Gamma, isFresh(f) \Rightarrow \mathcal{U}\{ invEv(\text{this}, e', f, m, \overline{e})\}\{ \ell := f \}[r] \Psi
\]

\[
\Gamma \Rightarrow \mathcal{U}[\ell = e'!m(\overline{e}); \ r] \Psi
\]
Semantic Evaluation of Asynchronous Method Call

\[
\text{val}_{\sigma}^{O,F}(\ell = e'!m(\overline{e})) = \{ pc \triangleright invEv_\sigma(O, \text{val}_{\sigma}^{O,F}(e'), f, m, \text{val}_{\sigma}^{O,F}(\overline{e})) \} \text{ if } \text{isFresh}(f), \text{method}(f) = m, pc \triangleright \tau \in \text{val}_{\sigma}^{O,F}(\ell = f) \}
\]

Sequent Rule for Asynchronous Method Call

\[
\Gamma, \text{isFresh}(f) \Rightarrow U\{ invEv(\text{this}, e', f, m, \overline{e})\}\{ \ell := f \}[r]\Psi \\
\Gamma \Rightarrow U[\ell = e'!m(\overline{e}); r]\Psi
\]
Semantic Evaluation of Asynchronous Method Call

\[
\text{val}_\sigma^{O, F}(\ell = e'!m(\overline{e})) = \{ pc \triangleright invEv_\sigma(O, \text{val}_\sigma^{O, F}(e'), f, m, \text{val}_\sigma^{O, F}(\overline{e})) \quad \text{isFresh}(f), \text{method}(f) = m, \ pc \triangleright \tau \in \text{val}_\sigma^{O, F}(\ell = f) \}
\]

Sequent Rule for Asynchronous Method Call

\[
\frac{\Gamma, \text{isFresh}(f) \Rightarrow \mathcal{U}\{ invEv(\text{this}, e', f, m, \overline{e}) \}\{ \ell := f \}[r]\Psi}{\Gamma \Rightarrow \mathcal{U}[\ell = e'!m(\overline{e}); \ r]\Psi}
\]
Semantic Evaluation of Asynchronous Method Call

\[
\text{val}_{\sigma}^{O,F}(\ell = e'!m(\overline{e})) = \{ pc \triangleright invEv_{\sigma}(O, \text{val}_{\sigma}^{O,F}(e'), f, m, \text{val}_{\sigma}^{O,F}(\overline{e})) \mid \\
\quad isFresh(f), \ \text{method}(f) = m, \ pc \triangleright \tau \in \text{val}_{\sigma}^{O,F}(\ell = f) \}\\
\]

Sequent Rule for Asynchronous Method Call

\[
\begin{align*}
\Gamma, isFresh(f) & \Rightarrow \mathcal{U}\{ invEv(\text{this}, e', f, m, \overline{e}) \}\{\ell := f\}[r]\Psi \\
\hline
\Gamma & \Rightarrow \mathcal{U}[\ell = e'!m(\overline{e}); \ r]\Psi
\end{align*}
\]
Asynchronous Method Call
Semantics vs. Calculus

Semantic Evaluation of Asynchronous Method Call

\[ \text{val}_{\sigma,F}^O(\ell = e'!m(e)) = \{ pc \triangleright invEv_\sigma(O, \text{val}_{\sigma,F}^O(e'), f, m, \text{val}_{\sigma,F}^O(e)) \} \quad | \quad \text{isFresh}(f), \ \text{method}(f) = m, \ pc \triangleright \tau \in \text{val}_{\sigma,F}^O(\ell = f) \] 

Sequent Rule for Asynchronous Method Call

\[
\frac{
\Gamma, \text{isFresh}(f) \Rightarrow \mathcal{U}\{ invEv(\text{this}, e', f, m, e) \}\{\ell := f\}[r]\Psi
}{
\Gamma \Rightarrow \mathcal{U}[\ell = e'!m(e); \ r]\Psi
}
\]

One-to-one correspondence between semantics and deduction rule!
Future Work

Calculus

We have a sequent calculus for local invariant reasoning

- Turn into calculus for global reasoning (Richard)
- Generalize invariant into contract-based reasoning (Eduard)
- Formally prove soundness, possibly completeness
- Implement as part of the ABS variant of KeY
Future Work

Calculus

We have a sequent calculus for local invariant reasoning

▶ Turn into calculus for global reasoning (Richard)
▶ Generalize invariant into contract-based reasoning (Eduard)
▶ Formally prove soundness, possibly completeness
▶ Implement as part of the ABS variant of KeY

Semantics

Apply semantic framework to other concurrent languages

▶ Related active object languages, e.g., MSR’s Orleans
▶ C-like concurrent languages with preemption
Did You Notice the Oulipian Constraint?

G. Perec