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

Semantics

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

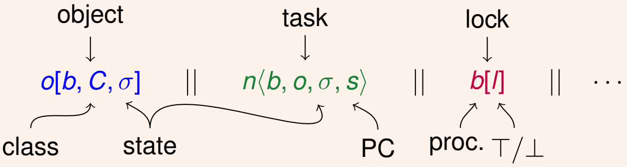
Given a program statement *s* and an initial state σ, the semantics of *s* is the set of all possible execution traces τ that are possible when *s* is started in σ

- 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

SOS: Structural Operational Semantics

Standard Approach in Programming Language Semantics

Reduction rules pattern match on runtime configurations ∼ states



object
task
lock
$o[b, C, \sigma] \quad \parallel \quad n\langle b, o, \sigma, s\rangle \quad \parallel \quad b[l] \quad \parallel \quad \cdots$
class
state
PC
proc. $\top/\bot$

Typical Reduction Rule: object creation:

$$\frac{n\langle b, o, \sigma, \texttt{T z = }\ \mathbf{new}\ \texttt{C(v);s}\rangle}{b'(\top) \parallel n'\langle b', o', \sigma'_{init}, s_{task}\rangle \parallel o'[b', \texttt{C}, \sigma_{init}] \parallel n\langle b, o, \sigma, \texttt{s}\{\texttt{z}/o'\}\rangle}$$

where b', o', n' new; $\overline{T\ f}; s'$ init block of $\texttt{C}$; $\sigma_{init} = \overline{T\ f}; s_{task} = s'\{\mathbf{this}/o'; \mathbf{suspend}\}$

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?

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! A least not for a complex, concurrent programming language

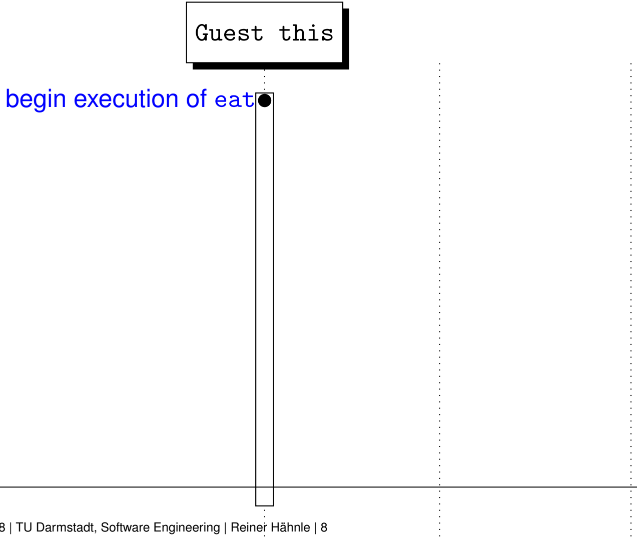
KeY, Why, Dafny, but even Dijkstra, Hoare	Calculus only
C, C++, Java, ABS	SOS only
Concurrent separation logic	Stephen Brookes' action traces

Starting Point

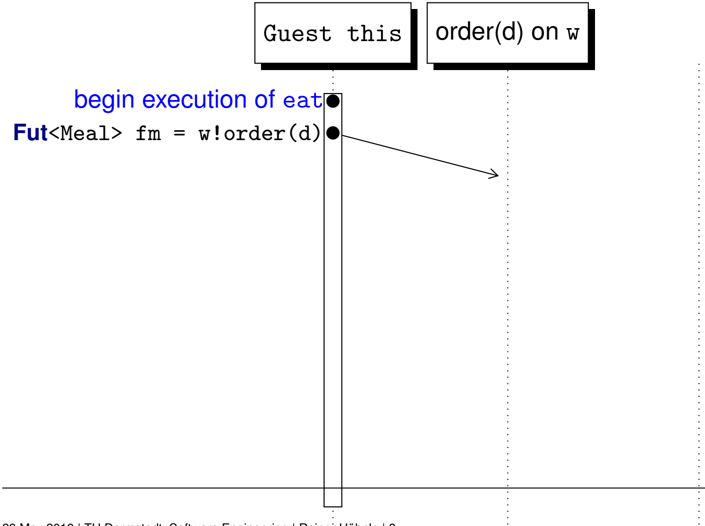
Model theoretic semantics for while language in:

Richard Bubel, Crystal Chang Din, Reiner Hähnle, Keiko Nakata.
A Dynamic Logic with Traces and Coinduction. TABLEAUX 2015: 307-322

ABS Communication Structure



ABS Communication Structure



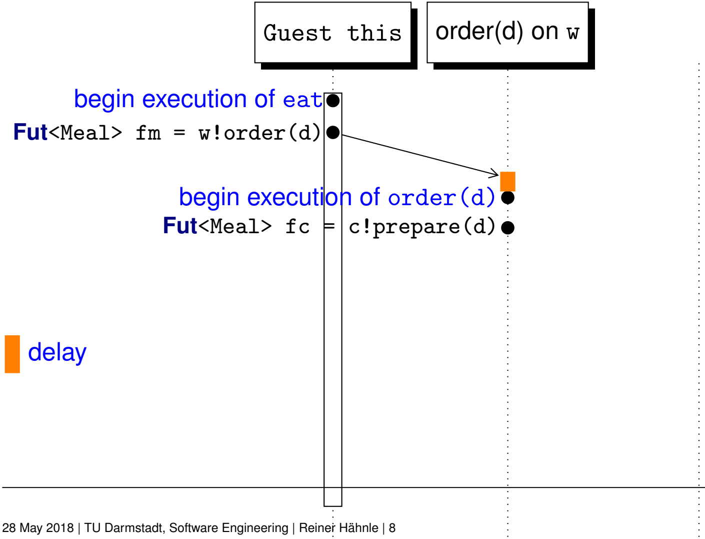
Guest this

order(d) on w

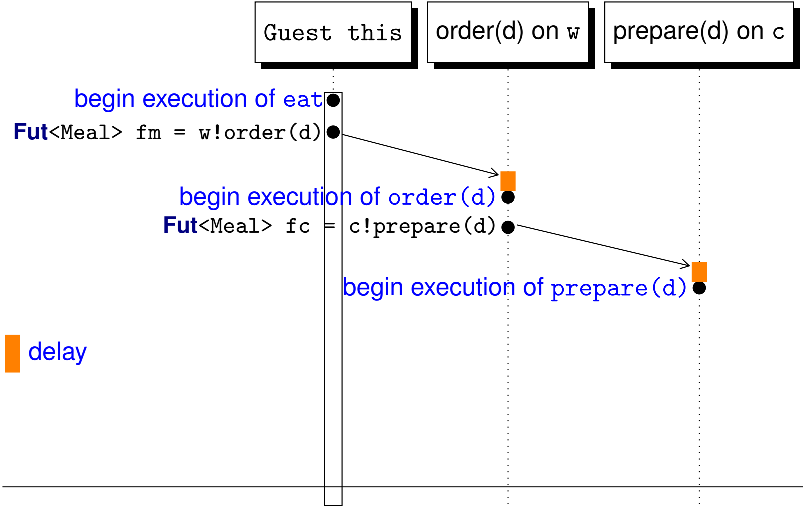
begin execution of eat

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

ABS Communication Structure

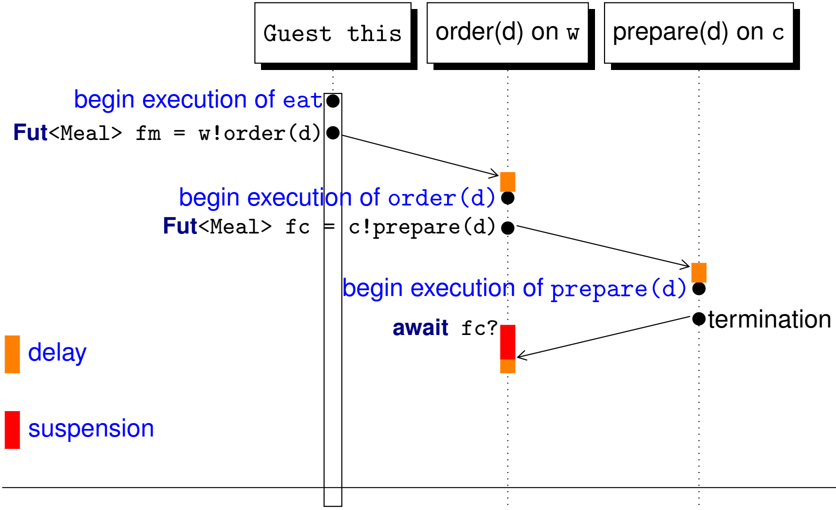

Guest this
order(d) on w
begin execution of eat
Fut<Meal> fm = w!order(d)
begin execution of order(d)
Fut<Meal> fc = c!prepare(d)
delay

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)
🟧 delay

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)

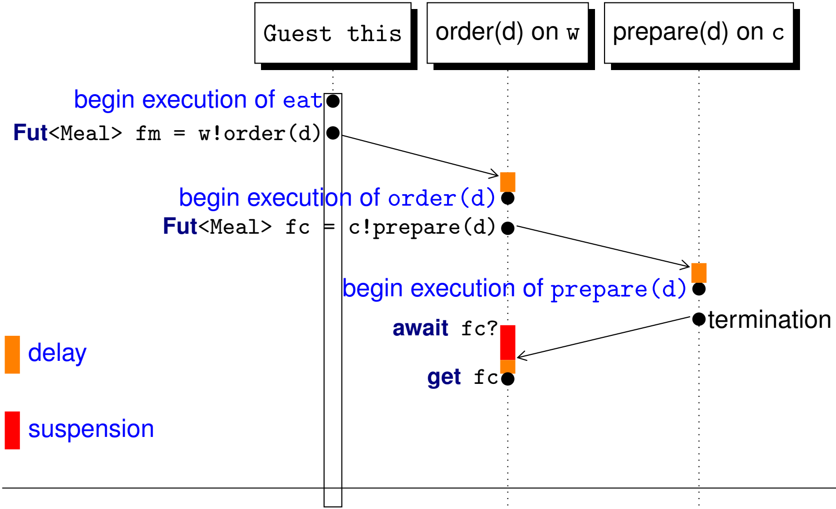
termination

await fc?

🟧 delay

🟥 suspension

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)

termination

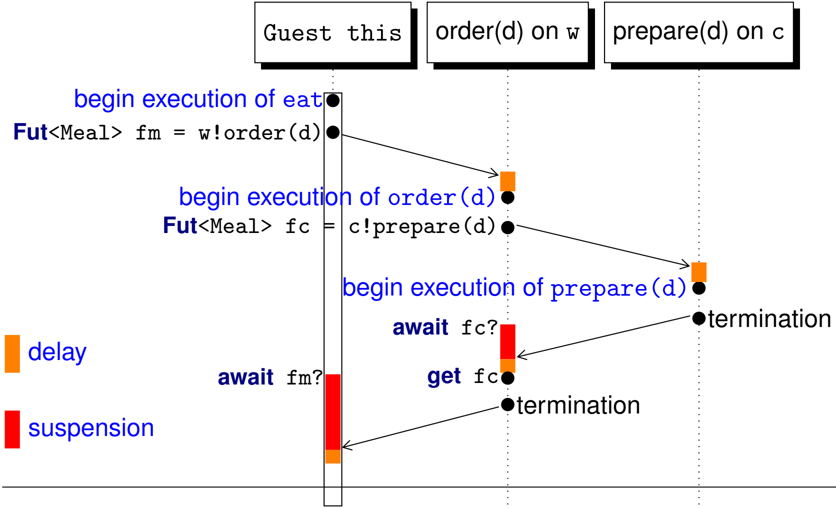
await fc?

get fc

🟧 delay

🟥 suspension

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)

termination

await fc?

get fc

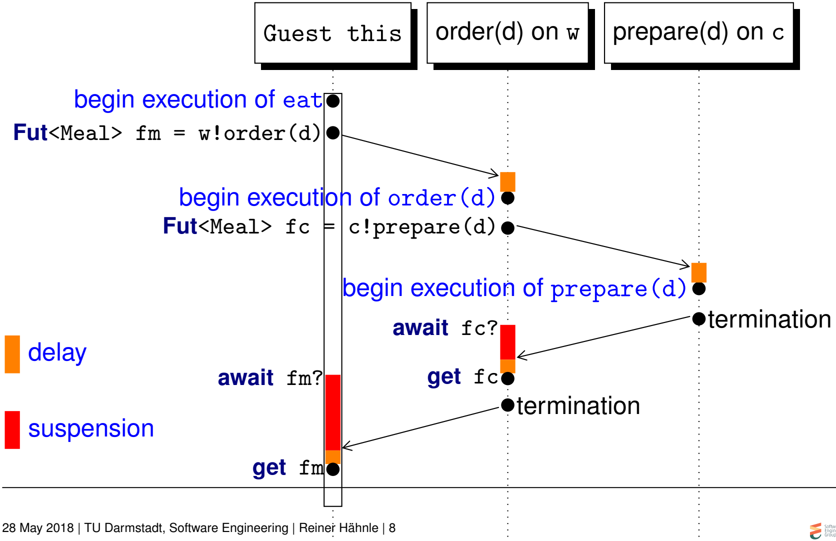
termination

await fm?

🟧 delay

🟥 suspension

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)

termination

await fc?

get fc

termination

await fm?

get fm

🟧 delay

🟥 suspension

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

A Denotational Semantics for ABS

Given an ABS sta e or infinite traces

. . . in the following ma

Local — for g), future **destiny**

Modular — evalu l on others’

Composable — obta ect-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 **this**, heap, future **destiny**

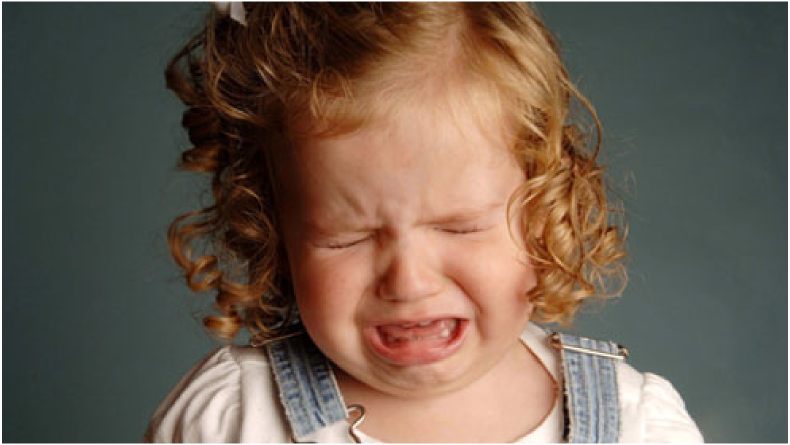
Modular — evaluation of one statement does not depend on others'

Composable — obtain global behavior by composition of object-local behavior

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

- 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

No!
Boldly Go where no Semantics has Gone Before

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

Abstract away from Unknowables during Semantic Evaluation

No!
Boldly Go where no Semantics has Gone Before

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

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

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

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

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

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

Symbolic, Conditioned Traces

Definition (Semantic Evaluation)

For each object O, future F, ABS statement s, symbolic state σ

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

is a semantic evaluation function yielding a set of symbolic traces starting in σ.

Symbolic, Conditioned Traces

Definition (Semantic Evaluation)

For each object O, future F, ABS statement s, symbolic state σ

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

is a semantic evaluation function yielding a set of symbolic traces starting in σ.

Definition (Path Condition, Symbolic Trace)

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

A (conditioned) symbolic trace has the form $pc \triangleright \tau$, where τ 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 σ

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

is a semantic evaluation function yielding a set of symbolic traces starting in σ.

Definition (Path Condition, Symbolic Trace)

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

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

Definition (Symbolic State)

A symbolic state is a function $\sigma : \mathcal{L} \to \mathrm{Exp}(\mathcal{L})$.

Symbolic Traces
Conventions, Notation, Example

- Assume $\sigma(\ell) \in \mathrm{Exp}(\mathcal{L})$ 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 τ
- Extend trace with single successor state: $\tau \curvearrowright \sigma$
- Lifting states to singleton traces: $\langle\sigma\rangle$
- Denote $\sigma(\ell) = e$ with $\ell \mapsto e$

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

- Empty trace denoted ε
- Concatenation of traces $\tau \cdot \omega$ (only defined if τ 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

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

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

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

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

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

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

$$\mathrm{val}_{\sigma}^{O,F}(\{T\,\ell = e; bs\}) =$$

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

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

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

$$\mathrm{val}_\sigma^{O,F}(\{T\ \ell = e; bs\}) = \\ pc \triangleright \omega \in \mathrm{val}_{\sigma'}^{O,F}(\{bs[\ell'/\ell]\}),\ \mathit{isFresh}(\ell')$$

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

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

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

$$\mathrm{val}_{\sigma}^{O,F}(\{T\ \ell = e; bs\}) = \begin{array}{l} \sigma' = \sigma[\ell' \mapsto \mathrm{val}_{\sigma}^{O,F}(e)], \\ pc \triangleright \omega \in \mathrm{val}_{\sigma'}^{O,F}(\{bs[\ell'/\ell]\}),\ isFresh(\ell') \end{array}$$

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

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

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

$$\begin{aligned}\mathrm{val}_{\sigma}^{O,F}(\{T\;\ell = e; bs\}) = \{pc \triangleright \langle\sigma\rangle \cdot \omega \mid \sigma' = \sigma[\ell' \mapsto \mathrm{val}_{\sigma}^{O,F}(e)],\\ pc \triangleright \omega \in \mathrm{val}_{\sigma'}^{O,F}(\{bs[\ell'/\ell]\}),\; isFresh(\ell')\}\end{aligned}$$

Local Semantics of ABS
Skip and Local Variable Assignment

$$\mathrm{val}_{\sigma}^{O,F}(\textbf{skip}) = \{\emptyset \triangleright \langle\sigma\rangle\}$$

$$\mathrm{val}_{\sigma}^{O,F}(\ell = e) = \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[\ell \mapsto \mathrm{val}_{\sigma}^{O,F}(e)]\}$$

Local Semantics of ABS
Skip and Local Variable Assignment

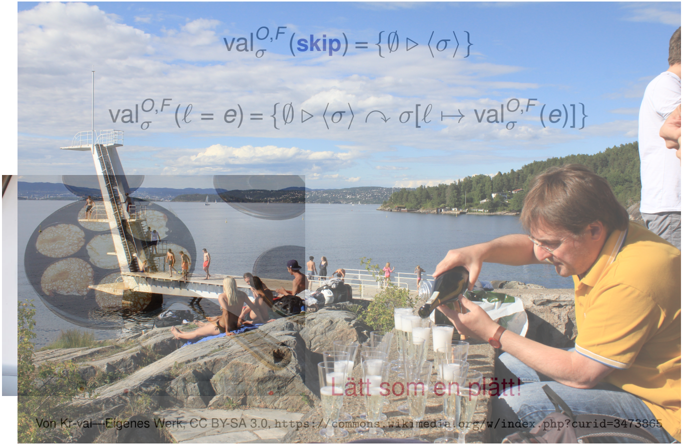
$$\mathsf{val}_\sigma^{O,F}(\mathbf{skip}) = \{\emptyset \triangleright \langle\sigma\rangle\}$$

$$\mathsf{val}_\sigma^{O,F}(\ell = e) = \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[\ell \mapsto \mathsf{val}_\sigma^{O,F}(e)]\}$$

Lätt som en plätt!

Von Kr-val—Eigenes Werk, CC BY-SA 3.0, `https://commons.wikimedia.org/w/index.php?curid=3473865`

Local Semantics of ABS
Skip and Local Variable Assignment



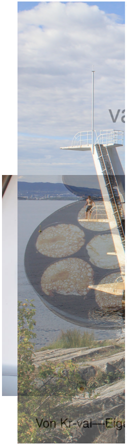
$$\mathrm{val}_\sigma^{O,F}(\textbf{skip}) = \{\emptyset \triangleright \langle\sigma\rangle\}$$

$$\mathrm{val}_\sigma^{O,F}(\ell = e) = \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[\ell \mapsto \mathrm{val}_\sigma^{O,F}(e)]\}$$

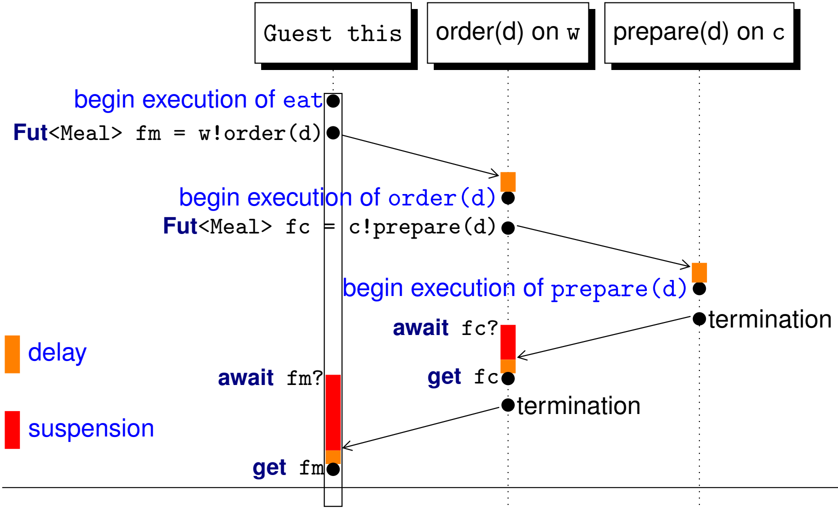
Lätt som en plätt!

Von Kr-val—Eigenes Werk, CC BY-SA 3.0, `https://commons.wikimedia.org/w/index.php?curid=3473865`

Local Semantics of ABS
Skip and Local Variable Assignment



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)

termination

await fc?

get fc

termination

await fm?

get fm

🟧 delay

🟥 suspension

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 σ inside a trace τ?

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

- 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 σ inside a trace τ?

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

- 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.

Local Semantics of ABS
Object Creation

```
class C(T̄ ā) implements I { ... }
ℓ = new C(ē); // create new object of class C
              // with class attribute arguments ē and assign to ℓ
```

Local Semantics of ABS
Object Creation

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

Object Initialization

- Wlog no initialization block

$$\mathrm{val}_{\sigma}^{O,F}(\ell = \mathbf{new}\ C\,(\overline{e})) =$$

Local Semantics of ABS
Object Creation

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

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 $\overline{v}$

$$\mathrm{val}_\sigma^{O,F}(\ell = \textbf{new}\ C\,(\overline{e})) =$$

$$\mathit{isFresh}(o),\ \mathit{class}(o) = C,\ \sigma' = C.\epsilon(o, \overline{v}) \circ \sigma,$$
$$\overline{v} = \mathrm{val}_\sigma^{O,F}(\overline{e})$$

Local Semantics of ABS
Object Creation

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

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 $\overline{v}$

$$\mathrm{val}_{\sigma}^{O,F}(\ell = \mathbf{new}\ C\,(\overline{e})) =$$

$$\mathit{isFresh}(o),\ \mathit{class}(o) = C,\ \sigma' = C.\epsilon(o, \overline{v}) \circ \sigma,$$
$$\mathit{pc} \triangleright \tau \in \mathrm{val}_{\sigma'}^{O,F}(\ell = o),\ \overline{v} = \mathrm{val}_{\sigma}^{O,F}(\overline{e})$$

Local Semantics of ABS
Object Creation

```
class C(T̄ ā) implements I { ... }
ℓ = new C(ē); // create new object of class C
              // with class attribute arguments ē and assign to ℓ
```

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 $\mathit{newEv}_\sigma(\texttt{this}, \texttt{newObject}, \texttt{attributeValues})$

$$\mathrm{val}_\sigma^{O,F}(\ell = \textbf{new}\ C\,(\overline{e})) = \{\mathit{pc} \triangleright \mathit{newEv}_\sigma(O, o, \overline{v}) \cdot \tau \mid \\ \mathit{isFresh}(o),\ \mathit{class}(o) = C,\ \sigma' = C.\epsilon(o, \overline{v}) \circ \sigma, \\ \mathit{pc} \triangleright \tau \in \mathrm{val}_{\sigma'}^{O,F}(\ell = o),\ \overline{v} = \mathrm{val}_\sigma^{O,F}(\overline{e})\}$$

Local Semantics of ABS
Asynchronous Method Call

ℓ = e'!$m(\overline{e})$; // *asynchronous call of* m *on* e' *with arguments* $\overline{e}$
// *assign result to* ℓ

Local Semantics of ABS
Asynchronous Method Call

$\ell\ =\ e'!m(\overline{e});$ $//\ \textit{asynchronous call of}\ m\ \textit{on}\ e'\ \textit{with arguments}\ \overline{e}$
$//\ \textit{assign result to}\ \ell$

Event Marker

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

Local Semantics of ABS
Asynchronous Method Call

$\ell \;=\; e'!m(\overline{e});$ // *asynchronous call of* m *on* e' *with arguments* $\overline{e}$
// *assign result to* ℓ

Event Marker

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

$$\mathrm{val}_\sigma^{O,F}(\ell = e'!m(\overline{e})) =$$

Local Semantics of ABS
Asynchronous Method Call

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

Event Marker

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

$$\mathrm{val}_\sigma^{O,F}(\ell = e'!m(\overline{e})) =$$

$$\mathit{isFresh}(f),\ \mathit{method}(f) = m,\ \mathit{pc} \triangleright \tau \in \mathrm{val}_\sigma^{O,F}(\ell = f)$$

Local Semantics of ABS
Asynchronous Method Call

$\ell\ =\ e'!m(\overline{e});$ // *asynchronous call of* m *on* e' *with arguments* $\overline{e}$
// *assign result to* ℓ

Event Marker

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

$$\mathrm{val}_\sigma^{O,F}(\ell = e'!m(\overline{e})) = \{pc \triangleright \mathit{invEv}_\sigma(O, \mathrm{val}_\sigma^{O,F}(e'), f, m, \mathrm{val}_\sigma^{O,F}(\overline{e}))\ \underline{**}\ \tau \mid \mathit{isFresh}(f),\ \mathit{method}(f) = m,\ pc \triangleright \tau \in \mathrm{val}_\sigma^{O,F}(\ell = f)\}$$

Local Semantics of ABS
Asynchronous Method Call

$\ell \;=\; e'!m(\overline{e});$ // asynchronous call of m on e' with arguments $\overline{e}$
// assign result to ℓ

Event Marker

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

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

What is "$\underline{**}$" ?

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

$$r; s$$

$$\underbrace{\sigma_0 \cdots \sigma_n}_{r} \qquad \underbrace{\sigma_n\, \sigma_{n+1} \cdots}_{s}$$

The “Chop” Constructor for Traces

Semantics of the Sequencing Statement in Terms of Traces

$$r; s$$

$$\underbrace{\sigma_0 \cdots \sigma_n}_{r} \quad \underline{**} \quad \underbrace{\sigma_n\,\sigma_{n+1} \cdots}_{s}$$

- $\tau \,\underline{**}\, \tau'$ is “chop” on traces: cut out one redundant state

The “Chop” Constructor for Traces

Semantics of the Sequencing Statement in Terms of Traces

$$r;s$$

$$||$$

$$\underbrace{\sigma_0 \cdots \sigma_n}_{r} \quad \underline{**} \quad \underbrace{\sigma_n\,\sigma_{n+1} \cdots}_{s}$$

- $\tau \; \underline{**} \; \tau'$ is “chop” on traces: cut out one redundant state

The “Chop” Constructor for Traces

Semantics of the Sequencing Statement in Terms of Traces

$$\overbrace{\sigma_0 \cdots \sigma_n \, \sigma_{n+1} \cdots}^{r;s}$$

$$||$$

$$\underbrace{\sigma_0 \cdots \sigma_n}_{r} \quad \underline{**} \quad \underbrace{\sigma_n \, \sigma_{n+1} \cdots}_{s}$$

- $\tau \,\underline{**}\, \tau'$ is “chop” on traces: cut out one redundant state
- If τ is infinite, returns τ, otherwise defined as above

The “Chop” Constructor for Traces

Semantics of the Sequencing Statement in Terms of Traces

$$\overbrace{\sigma_0 \cdots \sigma_n\,\sigma_{n+1} \cdots}^{r;s}$$

$$||$$

$$\underbrace{\sigma_0 \cdots \sigma_n}_{r} \quad \underline{**} \quad \underbrace{\sigma_n\,\sigma_{n+1} \cdots}_{s}$$

- $\tau \;\underline{**}\; \tau'$ is “chop” on traces: cut out one redundant state
- If τ is infinite, returns τ, otherwise defined as above
- Event lifting $ev_\sigma(\overline{v}) = \langle\sigma\rangle \curvearrowright ev(\overline{v}) \curvearrowright \sigma$: events are “choppable”

Local Semantics of ABS
Method Execution

Event Marker

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

$$\mathrm{val}_\sigma^{O,F}(C.m) =$$

Local Semantics of ABS
Method Execution

Event Marker

Invocation reaction event $invREv_\sigma(caller, callee, future, method, args)$

$$\mathrm{val}_\sigma^{O,F}(C.m) =$$

$$lookup(m, C) = T\ m(\overline{T}\ \overline{\ell'})\{s\},$$

Local Semantics of ABS
Method Execution

Event Marker

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

$$\mathrm{val}_\sigma^{O,F}(C.m) =$$

$$pc \triangleright \omega \in \mathrm{val}_\sigma^{O,F}(\{\overline{T}\ \overline{\ell'} = \overline{v_0}; s\}),\ \mathit{isFresh}(O', \overline{v_0})$$

$$\mathit{lookup}(m, C) = T\ m(\overline{T}\ \overline{\ell'})\{s\},$$

- 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 $\mathit{invREv}_\sigma(\mathit{caller}, \mathit{callee}, \mathit{future}, \mathit{method}, \mathit{args})$

$$
\begin{aligned}
\mathrm{val}_\sigma^{O,F}(C.m) = \{ pc \triangleright \mathit{invREv}_\sigma(O', O, F, m, \overline{v_0}) \underline{**}\, \omega \mid \\
pc \triangleright \omega \in \mathrm{val}_\sigma^{O,F}(\{\overline{T}\ \overline{\ell'} = \overline{v_0}; s\}),\ \mathit{isFresh}(O', \overline{v_0}) \\
\mathit{lookup}(m, C) = T\ m(\overline{T}\ \overline{\ell'})\{s\}, \}
\end{aligned}
$$

- 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 $\mathit{invREv}_\sigma(\mathit{caller}, \mathit{callee}, \mathit{future}, \mathit{method}, \mathit{args})$

$$\begin{aligned}\mathrm{val}_\sigma^{O,F}(C.m) = \{ & pc \triangleright \mathit{invREv}_\sigma(O', O, F, m, \overline{v_0}) \underline{**}\, \omega \mid \\ & pc \triangleright \omega \in \mathrm{val}_\sigma^{O,F}(\{\overline{T}\ \overline{\ell'} = \overline{v_0}; s\}),\ \mathit{isFresh}(O', \overline{v_0}) \\ & \mathit{lookup}(m, C) = T\ m(\overline{T}\ \overline{\ell'})\{s\}, \}\end{aligned}$$

- 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

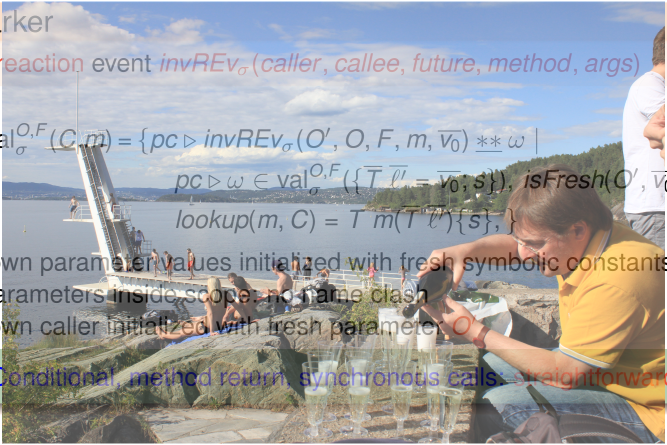
Event Marker

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

$$\begin{aligned}\mathrm{val}_\sigma^{O,F}(C.m) = \{ & pc \triangleright invREv_\sigma(O', O, F, m, \overline{v_0}) \mathbin{\underline{**}} \omega \mid \\ & pc \triangleright \omega \in \mathrm{val}_\sigma^{O,F}(\{\overline{T}\ \overline{\ell'} = \overline{v_0}; s\}),\ isFresh(O', \overline{v_0}) \\ & lookup(m, C) = T\ m(\overline{T}\ \overline{\ell'})\{s\}, \}\end{aligned}$$

- 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 ev d, args)

$\mathrm{val}_{\sigma}^{O,F}(C.m)$ $\mathit{resh}(O', \overline{v_0})$

- Unknown param constants $\overline{v_0}$
- Call parameters
- Unknown caller

Conditiona ghtforward

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

$$\mathrm{val}_{\sigma}^{O,F}(\textbf{suspend}) =$$

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

$$\mathrm{val}_{\sigma}^{O,F}(\textbf{suspend}) = \{\emptyset \triangleright \mathit{relEv}_{\sigma}(O) \cdot \mathit{relCont}(O, F, \textbf{skip})\}$$

- *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

$$\mathrm{val}_{\sigma}^{O,F}(\textbf{suspend}) = \{\emptyset \triangleright \mathit{relEv}_{\sigma}(O) \cdot \mathit{starve}(O)\} \;\cup\; \{\emptyset \triangleright \mathit{relEv}_{\sigma}(O) \cdot \mathit{relCont}(O, F, \textbf{skip})\}$$

- *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

Semantics of Sequential Composition — Continuation Passing Style

In a local semantics, sequential composition becomes tricky

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

Semantics of Sequential Composition — Continuation Passing Style

In a local semantics, sequential composition becomes tricky

$$
\begin{aligned}
&\mathrm{val}_{\sigma}^{O,F}(r;s) = \\
&\quad \{(pc_r \triangleright \tau_r) \underline{**} (pc_s \triangleright \omega_s) \mid pc_r \triangleright \tau_r \in \mathrm{val}_{\sigma}^{O,F}(r),\ pc_s \triangleright \omega_s \in \mathrm{val}_{\sigma'}^{O,F}(s), \\
&\qquad \text{where } \sigma' = \mathrm{last}(\tau_r) \text{ if } \tau_r \text{ is finite} \qquad \}
\end{aligned}
$$

Semantics of Sequential Composition — Continuation Passing Style

In a local semantics, sequential composition becomes tricky

- Execution of *r* might diverge or starve

$$
\begin{aligned}
&\mathrm{val}^{O,F}_{\sigma}(r;s) = \\
&\quad \{(pc_r \triangleright \tau_r) \;\underline{**}\; (pc_s \triangleright \omega_s) \mid pc_r \triangleright \tau_r \in \mathrm{val}^{O,F}_{\sigma}(r),\ pc_s \triangleright \omega_s \in \mathrm{val}^{O,F}_{\sigma'}(s), \\
&\qquad \text{where } \sigma' = \mathrm{last}(\tau_r) \text{ if } \tau_r \text{ is finite} \qquad\qquad \}
\end{aligned}
$$

Semantics of Sequential Composition — Continuation Passing Style

In a local semantics, sequential composition becomes tricky

- Execution of r might diverge or starve

$$
\begin{aligned}
&\mathrm{val}_{\sigma}^{O,F}(r;s) = \\
&\quad \{(pc_r \triangleright \tau_r) \;\underline{**}\; (pc_s \triangleright \omega_s) \mid pc_r \triangleright \tau_r \in \mathrm{val}_{\sigma}^{O,F}(r),\ pc_s \triangleright \omega_s \in \mathrm{val}_{\sigma'}^{O,F}(s), \\
&\qquad \text{where } \sigma' = \mathrm{last}(\tau_r) \text{ if } \tau_r \text{ is finite, arbitrary otherwise}\}
\end{aligned}
$$

- Non-termination, starving handled in definition of $\underline{**}$: throw away ω_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

$$
\begin{aligned}
&\mathrm{val}_{\sigma}^{O,F}(r;s) = \\
&\quad \{(pc_r \triangleright \tau_r) \,\underline{**}\, (pc_s \triangleright \omega_s) \mid pc_r \triangleright \tau_r \in \mathrm{val}_{\sigma}^{O,F}(r),\ pc_s \triangleright \omega_s \in \mathrm{val}_{\sigma'}^{O,F}(s), \\
&\qquad \text{where } \sigma' = \mathrm{last}(\tau_r) \text{ if } \tau_r \text{ is finite, arbitrary otherwise}\}
\end{aligned}
$$

- Non-termination, starving handled in definition of $\underline{**}$: throw away ω_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

$$
\begin{aligned}
&\mathrm{val}_{\sigma}^{O,F}(r;s) = \\
&\quad \{(pc_r \triangleright \tau_r)\ \underline{**}\ (pc_s \triangleright \omega_s) \mid pc_r \triangleright \tau_r \in \mathrm{val}_{\sigma}^{O,F}(r),\ pc_s \triangleright \omega_s \in \mathrm{val}_{\sigma'}^{O,F}(s), \\
&\qquad \text{where } \sigma' = \mathrm{last}(\tau_r) \text{ if } \tau_r \text{ is finite, arbitrary otherwise}\} \ \cup \\
&\quad \{pc_r \triangleright \tau_r \cdot relCont(O, F, r'; s) \mid pc_r \triangleright \tau_r \cdot relCont(O, F, r') \in \mathrm{val}_{\sigma}^{O,F}(r)\}
\end{aligned}
$$

- Non-termination, starving handled in definition of $\underline{**}$: throw away ω_s
- τ_r must end with continuation marker: compose with s

Semantic Evaluation of Sequential Composition Example

$$\mathrm{val}_{\sigma}^{O,F}(\textbf{suspend}; s) \quad \overset{?}{=}$$

Semantic Evaluation of Sequential Composition Example

$$\mathrm{val}_{\sigma}^{O,F}(\textbf{suspend}; s) \quad \overset{?}{=}$$

$$\begin{aligned}\mathrm{val}_{\sigma}^{O,F}(\textbf{suspend}) = {} & \{\emptyset \triangleright \mathit{relEv}_{\sigma}(O) \cdot \mathit{starve}(O)\} \cup \\ & \{\emptyset \triangleright \mathit{relEv}_{\sigma}(O) \cdot \mathit{relCont}(O, F, \textbf{skip})\}\end{aligned}$$

Semantic Evaluation of Sequential Composition Example

$$\mathrm{val}_{\sigma}^{O,F}(\textbf{suspend}; s) \overset{?}{=} \{\emptyset \triangleright \mathit{relEv}_{\sigma}(O) \cdot \mathit{starve}(O)\} \cup$$

$$\begin{aligned}\mathrm{val}_{\sigma}^{O,F}(\textbf{suspend}) = {} & \{\emptyset \triangleright \mathit{relEv}_{\sigma}(O) \cdot \mathit{starve}(O)\} \cup \\ & \{\emptyset \triangleright \mathit{relEv}_{\sigma}(O) \cdot \mathit{relCont}(O, F, \textbf{skip})\}\end{aligned}$$

Semantic Evaluation of Sequential Composition Example

$$\mathrm{val}_\sigma^{O,F}(\textbf{suspend}; s) \overset{?}{=} \begin{array}{l}\{\emptyset \triangleright \mathit{relEv}_\sigma(O) \cdot \mathit{starve}(O)\} \;\cup \\ \{\emptyset \triangleright \mathit{relEv}_\sigma(O) \cdot \mathit{relCont}(O, F, \textbf{skip}; s)\}\end{array}$$

$$\mathrm{val}_\sigma^{O,F}(\textbf{suspend}) = \begin{array}{l}\{\emptyset \triangleright \mathit{relEv}_\sigma(O) \cdot \mathit{starve}(O)\} \cup \\ \{\emptyset \triangleright \mathit{relEv}_\sigma(O) \cdot \mathit{relCont}(O, F, \textbf{skip})\}\end{array}$$

Computation of Local Traces
Example

```
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;
}
```

$$\mathrm{val}^{O,F}_{C.\epsilon(O)}(\texttt{C.n}) = \Big\{ \\ \Big\}$$

Computation of Local Traces
Example

```
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;
}
```

$\mathrm{val}^{O,F}_{\mathtt{C}.\epsilon(O)}(\mathtt{C.n}) = \{$

$\{(10 = 0)\} \triangleright \langle [O.i \mapsto v_i] \rangle \curvearrowright invREv(O', O, F, \mathtt{n}, \_) \curvearrowright \cdots$ infeasible path condition

$\}$

Computation of Local Traces
Example

```
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;
}
```

$$\mathsf{val}^{O,\,F}_{\mathsf{C}.\epsilon(O)}(\mathtt{C.n}) \;=\; \Big\{$$

$$\{(10 \neq 0)\} \triangleright \langle[O.i \mapsto v_i]\rangle \curvearrowright \mathit{invREv}(O', O, F, \mathtt{n}, \_) \curvearrowright \cdots$$

$$\curvearrowright \mathit{futEv}(O, F, v_i + 10) \curvearrowright [O.i \mapsto v_i,\; y' \mapsto v_i + 10,\; l' \mapsto f_0]$$

$$\Big\}$$

- Future f_0 in l already resolved at **await**: `n` runs to completion

Computation of Local Traces
Example

```
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;
}
```

$$\mathsf{val}^{O,F}_{C.\epsilon(O)}(\mathtt{C.n}) = \{$$

$$\{(10 \neq 0)\} \triangleright \langle [O.i \mapsto v_i] \rangle \curvearrowright \mathit{invREv}(O', O, F, \mathtt{n}, \_) \curvearrowright \cdots \mathit{starve}(O)$$

$$\}$$

- `n` starves at **await**

Computation of Local Traces
Example

```
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;
}
```

$$\mathrm{val}^{O,F}_{C.\epsilon(O)}(\texttt{C.n}) = \Big\{ \{(10 \neq 0)\} \rhd \langle [O.i \mapsto v_i] \rangle \curvearrowright \cdots \curvearrowright \mathit{relEv}(O, f_0) \curvearrowright [O.i \mapsto v_i,\, y' \mapsto 10,\, l' \mapsto f_0] \cdot \mathit{relCont}(O, F, \textbf{await}\ l'?;\ y' = \textbf{this}.i + y';\ \textbf{return}\ y';) \Big\}$$

- Release control at **await**: put remaining code in continuation

From Local to Global Traces

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 $\{\overline{T}\ \overline{\ell} = \overline{v}; s\}$ and ABS program P
produce all valid, concrete system traces

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

From Local to Global Traces

TECHNISCHE UNIVERSITÄT DARMSTADT

Goal

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

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

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

From Local to Global Traces

Goal

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

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

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

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

$$\mathcal{G} = \{\mathrm{val}^{O,\,F}_{\mathrm{C}.\epsilon(O)}(C.m) \mid \mathit{class}(O) = \mathrm{C},\ m \in \mathit{mtd}(\mathrm{C}),\ O \in \mathcal{O},\ F \in \mathcal{F},\ \mathrm{C} \in P\}$$

3. Pick an initial concrete trace with path condition true from $\mathcal{M}$ and extend it with suitable instances from $\mathcal{G}$, repeat

Producing Global System Traces

TECHNISCHE UNIVERSITÄT DARMSTADT

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

$$\frac{\text{Conditions on } sh,\ q}{sh,\ q \rightarrow sh',\ q'}$$

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

External Interleaving

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

$$\frac{\Omega \in q \quad \mathit{object}(\Omega) = O \quad pc \triangleright \tau \cdot \omega \in \Omega \;\Big|\; \text{get symbolic trace on an } O \text{ from pool } q}{sh, q \rightarrow sh \qquad\qquad , q'}$$

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

$$\frac{\Omega \in q \quad \mathit{object}(\Omega) = O \quad \mathit{pc} \triangleright \tau \cdot \omega \in \Omega \qquad \mathrm{last}(\mathit{sh}) = \sigma}{\mathit{sh}, q \to \mathit{sh} \qquad\qquad , q'}$$

get symbolic trace on an O from pool q
get final concrete state σ from sh

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

$\Omega \in q \quad \mathit{object}(\Omega) = O \quad \mathit{pc} \triangleright \tau \cdot \omega \in \Omega$	get symbolic trace on an *O* from pool *q*
$\mathrm{last}(\mathit{sh}) = \sigma$	get final concrete state σ from *sh*
$\tau \neq \varepsilon$	make some finite progress

$$\mathit{sh}, q \to \mathit{sh} \qquad\qquad\qquad , q'$$

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

$\Omega \in q \quad \mathit{object}(\Omega) = O \quad pc \rhd \tau \cdot \omega \in \Omega$	get symbolic trace on an O from pool q
$\mathrm{last}(sh) = \sigma$	get final concrete state σ from sh
$\tau \neq \varepsilon$	make some finite progress
$\omega \notin \{\varepsilon, \mathit{relCont}(O, \_, \_), \mathit{starve}(O)\}$	don’t finish execution on O

$$sh, q \to sh \qquad\qquad\qquad\qquad , q'$$

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

$\Omega \in q \quad \mathit{object}(\Omega) = O \quad \mathit{pc} \triangleright \tau \cdot \omega \in \Omega$	get symbolic trace on an O from pool q
$\mathrm{last}(\mathit{sh}) = \sigma$	get final concrete state σ from sh
$\tau \neq \varepsilon$	make some finite progress
$\omega \notin \{\varepsilon, \mathit{relCont}(O, \_, \_), \mathit{starve}(O)\}$	don't finish execution on O
$\mathit{pc}_\sigma = \mathrm{true} \qquad \mathit{wf}(\mathit{sh} \underline{**} \tau_\sigma)$	σ-instance of $\mathit{pc} \triangleright \tau$ feasible, well-formed

$$\mathit{sh}, q \to \mathit{sh} \underline{**} \tau_\sigma \qquad\qquad , q'$$

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

$\Omega \in q \quad \mathit{object}(\Omega) = O \quad \mathit{pc} \triangleright \tau \cdot \omega \in \Omega$	get symbolic trace on an O from pool q
$\mathrm{last}(\mathit{sh}) = \sigma$	get final concrete state σ from sh
$\tau \neq \varepsilon$	make some finite progress
$\omega \notin \{\varepsilon, \mathit{relCont}(O, \_, \_), \mathit{starve}(O)\}$	don't finish execution on O
$\mathit{pc}_\sigma = \mathrm{true} \qquad \mathit{wf}(\mathit{sh} \underline{**} \tau_\sigma)$	σ-instance of $\mathit{pc} \triangleright \tau$ feasible, well-formed
$q' = q \setminus \Omega \cup \{\emptyset \triangleright \mathit{ilREv}_{\mathrm{last}(\tau)}(O) \cdot \omega\}$	update pool, insert interleaving events

$$\mathit{sh}, q \rightarrow \mathit{sh} \underline{**} \tau_\sigma \underline{**} \mathit{ilEv}_{\mathrm{last}(\tau_\sigma)}(O), q'$$

Other Global Trace Composition Rules

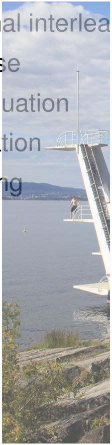
1. External interleaving
2. Release
3. Continuation
4. Starvation
5. Blocking

Other Global Trace Composition Rules

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

Other Global Trace Composition Rules

1. External interlea
2. Release
3. Continuation
4. Starvation
5. Blocking



Well-Formed Traces

Well-Formedness

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

- 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: $\mathit{wf}(\mathit{sh} \mathbin{\underline{**}} \tau_\sigma)$

- 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

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

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, and blocking

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, 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

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, 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

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, 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

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, starvati [illegible]
- Global behavior by instantia [illegible] rleaving
  - Can characterize concurr [illegible] medness by way of dual events
  - Separation of concerns: c [illegible] t structure

One evaluation rule per statement, five global rules

Trace Formulas

Symbolic Trace Formula

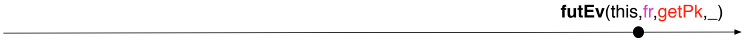
An (abstract) symbolic trace formula evaluates to a possibly infinite set of traces

Trace Formulas

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,”

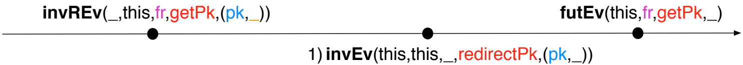


Trace Formulas

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”



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

futEv(this,fr,getPk,_)

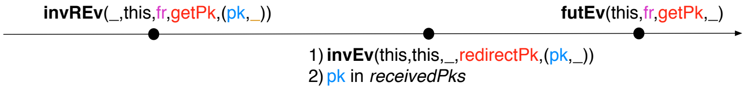
1) **invEv**(this,this,_,redirectPk,(pk,_))

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”



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

futEv(this,fr,getPk,_)

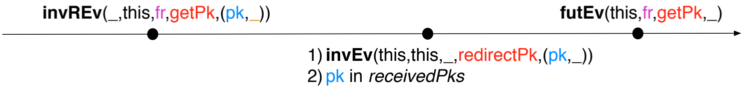
1) **invEv**(this,this,_,redirectPk,(pk,_))
2) pk in *receivedPks*

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”



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

futEv(this,fr,getPk,_)

1) **invEv**(this,this,_,redirectPk,(pk,_))
2) pk in *receivedPks*

Corresponding symbolic trace formula:

$$\omega_{fr,pk}\Big(\textbf{invREv}(\_,\textbf{this},fr,getPk,(pk,\_)) \ll \textbf{invEv}(\textbf{this},\textbf{this},\_,redirectPk,(pk,\_)) \ll \textbf{futEv}(\textbf{this},fr,getPk,\_) \\ \lor \\ \textbf{invREv}(\_,\textbf{this},fr,getPk,(pk,\_)) \ll \lceil pk \in receivedPks \rceil \ll \textbf{futEv}(\textbf{this},fr,getPk,\_)\Big)$$

ABS Program Logic

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 Ψ a trace modality formula, then $[\![s]\!]\Psi$ is a trace modality formula.
3. If $\{u\}$ is an update and Ψ a trace modality formula, then $\{u\}\Psi$ is a trace modality formula.

   Updates $\{\ell := \mathit{exp}\}$ or $\{\mathit{ev}(\overline{e})\}$ record state changes effected by assignments or the occurrence of communication events.

Semantics of Trace Modality Formulas

Definition (Evaluation of Trace Modality Formula)

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

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

“Any trace of s that extends τ is contained in Ψ”

Semantics of Trace Modality Formulas

TECHNISCHE UNIVERSITÄT DARMSTADT

Definition (Evaluation of Trace Modality Formula)

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

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

“Any trace of s that extends τ is contained in Ψ”

With a semantics for trace modality formulas, we can start to design a calculus . . .

Asynchronous Method Call
Semantics vs. Calculus

Semantic Evaluation of Asynchronous Method Call

$$\mathrm{val}_\sigma^{O,F}(\ell = e'!m(\overline{e})) = \{pc \triangleright \mathit{invEv}_\sigma(O, \mathrm{val}_\sigma^{O,F}(e'), f, m, \mathrm{val}_\sigma^{O,F}(\overline{e})) \mathbin{\underline{**}} \tau \mid \\ \mathit{isFresh}(f),\ \mathit{method}(f) = m,\ pc \triangleright \tau \in \mathrm{val}_\sigma^{O,F}(\ell = f)\}$$

Asynchronous Method Call
Semantics vs. Calculus

Semantic Evaluation of Asynchronous Method Call

$$\mathrm{val}_\sigma^{O,F}(\ell = e'!m(\overline{e})) = \{pc \triangleright \mathit{invEv}_\sigma(O, \mathrm{val}_\sigma^{O,F}(e'), f, m, \mathrm{val}_\sigma^{O,F}(\overline{e}))\ \underline{**}\ \tau \mid \\ \mathit{isFresh}(f),\ \mathit{method}(f) = m,\ pc \triangleright \tau \in \mathrm{val}_\sigma^{O,F}(\ell = f)\}$$

Sequent Rule for Asynchronous Method Call

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

Asynchronous Method Call
Semantics vs. Calculus

Semantic Evaluation of Asynchronous Method Call

$$\mathrm{val}^{O,F}_{\sigma}(\ell = e'!m(\overline{e})) = \{pc \triangleright \mathit{invEv}_{\sigma}(O, \mathrm{val}^{O,F}_{\sigma}(e'), f, m, \mathrm{val}^{O,F}_{\sigma}(\overline{e})) \mathbin{\underline{**}} \tau \mid \mathit{isFresh}(f),\ \mathit{method}(f) = m,\ pc \triangleright \tau \in \mathrm{val}^{O,F}_{\sigma}(\ell = f)\}$$

Sequent Rule for Asynchronous Method Call

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

Asynchronous Method Call
Semantics vs. Calculus

Semantic Evaluation of Asynchronous Method Call

$$\mathrm{val}_\sigma^{O,F}(\ell = e'!m(\overline{e})) = \{pc \triangleright \mathit{invEv}_\sigma(O, \mathrm{val}_\sigma^{O,F}(e'), f, m, \mathrm{val}_\sigma^{O,F}(\overline{e}))\ \underline{**}\ \tau \mid \mathit{isFresh}(f),\ \mathit{method}(f) = m,\ pc \triangleright \tau \in \mathrm{val}_\sigma^{O,F}(\ell = f)\}$$

Sequent Rule for Asynchronous Method Call

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

Asynchronous Method Call
Semantics vs. Calculus

TECHNISCHE UNIVERSITÄT DARMSTADT

Semantic Evaluation of Asynchronous Method Call

$$\mathrm{val}_\sigma^{O,F}(\ell = e'!m(\overline{e})) = \{pc \triangleright \mathit{invEv}_\sigma(O, \mathrm{val}_\sigma^{O,F}(e'), f, m, \mathrm{val}_\sigma^{O,F}(\overline{e})) \mathbin{\underline{**}} \tau \mid \mathit{isFresh}(f),\ \mathit{method}(f) = m,\ pc \triangleright \tau \in \mathrm{val}_\sigma^{O,F}(\ell = f)\}$$

Sequent Rule for Asynchronous Method Call

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

Asynchronous Method Call
Semantics vs. Calculus

Semantic Evaluation of Asynchronous Method Call

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

Sequent Rule for Asynchronous Method Call

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

Asynchronous Method Call
Semantics vs. Calculus

Semantic Evaluation of Asynchronous Method Call

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

Sequent Rule for Asynchronous Method Call

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

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