Heap-Abstraction for a Multithreaded Object-Oriented Calculus

Erika Ábrahám    Martin Steffen

RWTH Aachen University

University of Oslo
1. Full abstractness

2. Classes and legal traces: realizability

3. Classes and observability: closure conditions

4. Conclusion
The talk is about programming languages with...

- classes, object creation
- no direct access to instance variables
- communication via synchronous method calls
- multithreading, thread creation
- reentrant monitors
- inheritance
Starting point

**Component** = “program fragment” = “open program”

**Environment** = “context” = “observer”

- Environment $c[\_\_]:$ “program with a hole”
- Filling the hole with a component $C$: closed program $c[C]$
Starting point

Component = “program fragment” = “open program”
Environment = “context” = “observer”

- Environment $C[\_\]$: “program with a hole”
- Filling the hole with a component $C$: closed program $C[C]$
Starting point

Questions:

- For verification: What is the semantics of a component?
- For component exchange: When do two components behave “equivalently”?

Or more precise:

What is observable about an open class-based, object-oriented, multithreaded program?

This question can be answered by defining a fully-abstract semantics.
Starting point

Questions:

- For verification: What is the semantics of a component?
- For component exchange: When do two components behave “equivalently”?

Or more precise:

What is observable about an open class-based, object-oriented, multithreaded program?

This question can be answered by defining a fully-abstract semantics.
Starting point

Questions:
- For verification: What is the semantics of a component?
- For component exchange: When do two components behave “equivalently”?

Or more precise:

What is observable about an open class-based, object-oriented, multithreaded program?

This question can be answered by defining a fully-abstract semantics.
Full abstraction

- Natural definition of equivalence of program fragments.
- **Comparison** between two semantics, resp. two implied notions of equality/preorder.

(1) **Reference** semantics: Notion of observation.
Given a closed program \( P \): \( \mathcal{O}(P) = \text{observations} \)
⇒ observational equivalence:

\[
C_1 \equiv_{\text{obs}} C_2 \iff \forall C. \mathcal{O}(C[C_1]) = \mathcal{O}(C[C_2])
\]

(2) The 2nd semantics is **fully abstract** wrt. the reference semantics if it is
- neither too abstract = **sound**
- nor too concrete = **complete**

Given a denotational semantics \( \llbracket - \rrbracket_D \), resp. the implied equality \( \equiv_D \)
⇒ \( \equiv_D \) is fully abstract wrt. \( \equiv_{\text{obs}} \):

\[
\equiv_{\text{obs}} = \equiv_D
\]
Full abstractness in an object-based concurrent setting

- Jeffrey, Rathke: For the concurrent \( \nu \)-calculus
- **Component** = set of parallelly “running” objects + threads
- **Observable**: Message exchange at the boundary

\[ \Rightarrow \] Fully abstract observable behavior = communication traces of the labels of the operational semantics

- Pretty simple **observational** notion:

  *Compose a component with an observer, let it run and see, whether the observer reports success by calling a specific context method ("o.success() ").*

- **May-testing** preorder: \( C_1 \sqsubseteq_{\text{may}} C_2 \) iff for all observers \( O \), if \( C_1 || O \) is successful, then so is \( C_2 || O \).
Notion of observation: Reference semantics

// component
public class C {
    public static void main(String[] arg) {
        O x = new O();
        x.m(42);
    }
}

Notion of observation: Reference semantics

// component
public class C {
    public static void main(String[] arg) {
        O x = new O();
        x.m(42);
    }
}

// external observer
class O {
    public void m(int x) {
        <some code>;
        System.out.println("success");
    }
}
Completeness: Line of argument

- **Goal:** if $C_1 \sqsubseteq_{\text{may}} C_2$, then $C_1 \sqsubseteq_{\text{trace}} C_2$

- So, given a legal trace $s \in \llbracket C_1 \rrbracket_{\text{trace}}$, do
  
  - construct a complementary context $C_{\bar{s}}$
  
  - composition: program + context may do the observation

  $$C_{\bar{s}}[C_1] \rightarrow^* success$$

  - observational equivalence: $C_2$ may do that, too:

  $$C_{\bar{s}}[C_2] \rightarrow^* success$$

  - decomposition: $s \in \llbracket C_2 \rrbracket_{\text{trace}}$

$\implies$ Problems for completeness (apart from technicalities):

1. **definability** ⇒ what are realizable/legal traces?
2. what can be observed/distinguished?
1. Full abstractness

2. Classes and legal traces: realizability

3. Classes and observability: closure conditions

4. Conclusion
Classes and full abstraction

- **Open semantics for objects** (based on may testing/observational equivalence): in principle straightforward and understood

  ⇒ Corresponding semantics is “traces” as interface interactions (messages, method calls and returns)

  What is the semantical import of classes?

Main issue:
- Interface separates component and observer classes
- Classes are generators of object (via `new`)
- Component instantiates observer class
  - Instance: part of the observer
  - Reference to it: kept at the component as interface interaction
Classes and full abstraction

- **Open semantics for objects** (based on may testing/observational equivalence): in principle straightforward and understood

  → Corresponding semantics is “traces” as interface interactions (messages, method calls and returns)

  \[
  \text{What is the semantical import of classes?}
  \]

- **Main issue:**
  - Interface separates component and observer classes
  - Classes are generators of object (via `new`)
  - Component instantiates observer class
    - ★ Instance: part of the observer
    - ★ Reference to it: kept at the component

  → Instantiation as interface interaction
Problems to tackle for fully abstract semantics

- Semantics: message interchange at the interface
- But: environment is absent/arbitrary
Problems to tackle for fully abstract semantics

- Semantics: message interchange at the interface
- But: environment is absent/arbitrary

Labels:

\[
\begin{align*}
\gamma & ::= \ n\langle \text{call } o.m(\vec{v}) \rangle \mid n\langle \text{return}(v) \rangle \\
& \quad \mid \langle \text{spawn } n \text{ of } c(\vec{v}) \rangle \mid \nu(n:T).\gamma \\
a & ::= \gamma? \mid \gamma!
\end{align*}
\]

basic labels

receive and send labels
Problems to tackle for fully abstract semantics

- Semantics: message interchange at the interface
- But: environment is absent/arbitrary

⇒ Does this mean: environment behavior arbitrary?
Problems to tackle for fully abstract semantics

- Semantics: message interchange at the interface
- But: environment is absent/arbitrary

⇒ Does this mean: environment behavior arbitrary?

- well, depends . . . does “arbitrary trace” mean ∈ Label* ?
Problems to tackle for fully abstract semantics

- Semantics: message interchange at the interface
- But: environment is absent/arbitrary

⇒ Does this mean: environment behavior arbitrary?

- well, depends . . . does “arbitrary trace” mean ∈ Label* ?
- We know $P\parallel O$ is a program of the language
  - well-formed
  - well-typed
  - class-structured
Problems to tackle for fully abstract semantics

- Semantics: message interchange at the interface
- But: environment is absent/arbitrary

⇒ Does this mean: environment behavior arbitrary?

- well, depends . . . does “arbitrary trace” mean $\in Label^*$ ?
- We know $P\|O$ is a program of the language
  - well-formed
  - well-typed
  - class-structured

Environment is arbitrary but realizable
Open semantics

- **Operational description:**

- **Assumption/commitment formulation**

  \[ \text{Ass} \vdash C : \text{Comm} \quad \Rightarrow \quad \text{Ass} \vdash \acute{C} : \acute{\text{Comm}} \]

- **Interface:** 3 orthogonal abstractions:
  - static abstraction: **type system**
  - abstraction of the **stack** structure of thread(s)
  - dynamic abstraction of the **heap topology**
1. Static abstraction: Type system

```java
// component
public class P{
    public void m(C x){
        ...
    }
}
```

E.g.: Method m of o:P must have one parameter of type C.

⇝ Traces

```
... n<call o.m(o')>? ...
```

with \( o, o' : P \) are not realizable.
2. Abstraction of the stack structure

E.g.:

- Calls and returns of a thread must occur in a nested fashion.

\[ n \langle \text{call o.m(...)} \rangle? \]
\[ n \langle \text{call o'.m(...)} \rangle? \]

are not realizable.
3. Dynamic abstraction of the heap topology

```java
// component
public class P {
    ...
    public void m() {
        C x = new C();
        C y = x.m();
    }
}
```

Is a trace

```plaintext
... \( \nu(o_2 : C).n \langle \text{call } o_2.m() \rangle ! \)
\( \nu(o_3 : C).n' \langle \text{call } o_3.m() \rangle ! \)
\( n' \langle \text{return}(o_2) \rangle ? \)
...
```

realizable?
Dynamic heap abstraction example

Component

\[ P \]

\[ o_1 \text{ creates } o_2 \]
\[ o_1 \text{ calls } o_2.m() \]

\[ o_1 \]

Environment

\[ C \]

\[ o_2 \]
Dynamic heap abstraction: example

Component

$P$

$o_1$ creates $o_2$

$o_1$ calls $o_2.m()$

$o_1$ creates $o_3$

$o_1$ calls $o_3.m()$

Environment

$C$

$o_2$

$o_3$
Dynamic heap abstraction: example

Component

$P$

$O_1$

$O_2$

$O_3$

Environment

$C$

$O_2$

$O_3$

$O_1$ creates $O_2$

$O_1$ calls $O_2.m()$

$O_1$ creates $O_3$

$O_1$ calls $O_3.m()$

$O_3$ returns $O_2$

$O_2$ and $O_3$ cannot “know” each other!
Dynamic heap abstraction: example

Component

$P$

$O_1$

$O_2$

$O_3$

$O_3$ returns $O_2$

$O_1$ creates $O_2$

$O_1$ calls $O_2.m()$

$O_1$ creates $O_3$

$O_1$ calls $O_3.m'(O_2)$

Environment

$C$

merging!

$O_2$

$O_3$
Dynamic heap abstraction: example

Component

$P$

$o_1$ creates $o_2$

$o_1$ calls $o_2.m()$

$o_1$ creates $o_3$

$o_1$ calls $o_3.m'(o_2)$

$o_2$ returns $o_3$

Environment

$C$

$o_2$

merging!

$o_3$
Dynamic aspects of cliques

- We have seen: cliques can merge
- Assumption: names are never forgotten
  ⇒ cliques never fall apart again
- Clique evolution represents a tree:
Open semantics and heap abstraction

- Exact interface behavior
  ⇒ Abstraction of the heap topology necessary
- Keep book about “who has been told what”:

\[ \Delta; E_\Delta \vdash C : \Theta; E_\Theta \]

- Assumption context: \( E_\Delta \subseteq \Delta \times (\Delta + \Theta) = \text{pairs of objects} \)
- Written \( o_1 \leftrightarrow o_2 \):
- Worst case: equational theory implied by \( E_\Delta \)

\[ o_1, o_2 \in \Delta : \quad E_\Delta \vdash o_1 \Leftrightarrow o_2 \]

\[ o_1, \in \Delta, \ o_2 \in \Theta : \quad E_\Delta \vdash o_1 \Leftrightarrow; \Leftrightarrow o_2 \]
Dynamic heap abstraction

- E.g., sending $o_1$ to $o_2$, adds $o_1 \leftrightarrow o_2$ to the equations
- outgoing call
  - both caller and callee are known
  - $a = n\langle \text{call } o_{\text{callee}} \cdot l(\vec{v}) \rangle !$

\[
\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{a} \Delta'; E'_\Delta \vdash \hat{C} : \hat{\Theta}; E'_\Theta
\]

- assumption update: $\hat{E}_\Delta = E_\Delta + o_{\text{callee}} \leftrightarrow \vec{v}$
Dynamic heap abstraction

- E.g., sending $o_1$ to $o_2$, adds $o_1 \leftarrow o_2$ to the equations
- outgoing call
  - both caller and callee are known
  - $a = n\langle call \ o_{callee}.l(\vec{v})\rangle$

\[ \Delta; E_{\Delta} \vdash C : \Theta; E_{\Theta} \xrightarrow{a} \Delta; E_{\Delta} \vdash C : \Theta; E_{\Theta} \]

- assumption update: $\dot{E}_\Delta = E_\Delta + o_{callee} \leftarrow \vec{v}$

- incoming call
  - only callee is known, caller is guessed
  - $a = n\langle call \ o_{callee}.l(\vec{v})\rangle$

\[ \Delta; E_{\Delta} \vdash C : \Theta; E_{\Theta} \xrightarrow{a} \Delta; E_{\Delta} \vdash C : \Theta; E_{\Theta} \]

- assumption check: $\dot{E}_\Delta \vdash o_{caller} \leftarrow \vec{v}$
Simplified rule for incoming call

\[ a = n\langle call \; o_r.l(\vec{v})\rangle? \]

update contexts: \( \Theta; \hat{E}_\Theta = \Theta; E_\Theta + o_r \leftrightarrow \vec{v}, n \)

check context: \( \hat{\Delta}; \hat{E}_\Delta \vdash o_s \leftrightarrow \vec{v}, o_r : \Theta \)

\[ \Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{a} \hat{\Delta}; \hat{E}_\Delta \vdash \hat{C} : \hat{\Theta}; \hat{E}_\Theta \]

\text{CALLI}
Where are we?

Open semantics in the presence of classes

- static abstraction of type system
- abstraction of the stack structure
- abstraction of heap topology
- formalized in some “object calculus”

But we are still not ready...
1. Full abstractness

2. Classes and legal traces: realizability

3. Classes and observability: closure conditions

4. Conclusion
Only one success report

- The observer is itself divided into cliques
- But: only one reports success
Order of events: Swapping

- separate observer cliques cannot cooperate

⇒ order of interaction not globally visible (but: take care of merging)
Deterministic traces: Replay

- More than one instance of a class in one trace
- Different instances of same class behave “the same”
- Equivalent stimulus/input history $\Rightarrow$ equivalent reaction

Example:

$$\nu(o:c)n\langle\text{call } o.l()\rangle? n\langle\text{return}(5)\rangle! \nu(o':c)n\langle\text{call } o'.l()\rangle? n\langle\text{return}(7)\rangle!$$
Deterministic traces: Replay

- More than one instance of a class in one trace
- Different instances of same class behave “the same”
- Equivalent stimulus/input history $\Rightarrow$ equivalent reaction
- Example:

$$\nu(o:c)n\langle call \ o.\!l()\rangle? \ n\langle \text{return}(5)\rangle! \ \nu(o':c)n\langle call \ o'.\!l()\rangle? \ n\langle \text{return}(7)\rangle!$$

$$\nu(o:c)n\langle call \ o.\!l(\text{true})\rangle? \ n\langle \text{return}(5)\rangle!$$

$$\nu(o':c)n\langle call \ o'.\!l(\text{false})\rangle? \ n\langle \text{return}(7)\rangle!$$
Semantics

$$\Xi_0 \vdash C_1 \sqsubseteq_{\text{trace}} C_2,$$

if for all deterministic $$\Xi_0 \vdash C_1 \xrightarrow{t}$$, there exists a deterministic $$\Xi_0 \vdash C_2 \xrightarrow{s}$$ such that $$\Xi_0 \vdash s \cong_\Delta t$$.

- $$\cong_\Delta$$: swapping and replay
- claim: $$\cong_\Delta$$ sound and complete wrt. observational preorder
1. Full abstractness

2. Classes and legal traces: realizability

3. Classes and observability: closure conditions

4. Conclusion
What I didn’t mention

- "determinism": 2 “identical” (up-to name) ⇒ same reaction
- treatment of “cross-border” instantiation:
  - instantiation itself is not visible
  - “lazy instantiation”
  - guessing connectivity also for instances the “other side” instantiated in the component (and vice versa)
- **caller** identity must ultimately be ignored
- coding issues
- . . .
Conclusions

- Fully abstract semantics for an
  - OO,
  - class-based,
  - multithreaded language.

Abstractions:
- type system
- stack structure
- heap topology
- closure conditions

Extensions:
- sequential language
- thread classes
- monitors
- subtyping (and subclassing), cloning, ...
- (fully) compositional semantics