Inheritance and Observability

Erika Ábrahám, Mai Thuong Tran, and Martin Steffen

RWTH Aachen, Germany and University of Oslo, Norway

Nordic Workshop on Programming Theory, NWPT2011,
Västerås, Sweden, October 26-28, 2011

What are we dealing with?

- Class-based object-oriented multi-threaded programming languages with inheritance

What's the observable behavior of open programs in the presence of inheritance?

- Why important?
  - verification
  - black-box testing
  - compositionality, replacement, full abstraction

$\Longrightarrow$ Easy question, difficult answer

$\Longrightarrow$ Open semantics.

What are we dealing with?

- Class-based object-oriented multi-threaded programming languages with inheritance

What's the observable behavior of open programs in the presence of inheritance?

- Why important?
  - verification
  - black-box testing
  - compositionality, replacement, full abstraction

$\Longrightarrow$ Easy question, difficult answer

$\Longrightarrow$ Open semantics.

What are we dealing with?

- Class-based object-oriented multi-threaded programming languages with inheritance

What's the observable behavior of open programs in the presence of inheritance?

- Why important?
  - verification
  - black-box testing
  - compositionality, replacement, full abstraction

$\Longrightarrow$ Easy question, difficult answer

$\Longrightarrow$ Open semantics.

What are we dealing with?

- Class-based object-oriented multi-threaded programming languages with inheritance

What's the observable behavior of open programs in the presence of inheritance?

- Why important?
  - verification
  - black-box testing
  - compositionality, replacement, full abstraction

$\Longrightarrow$ Easy question, difficult answer

$\Longrightarrow$ Open semantics.

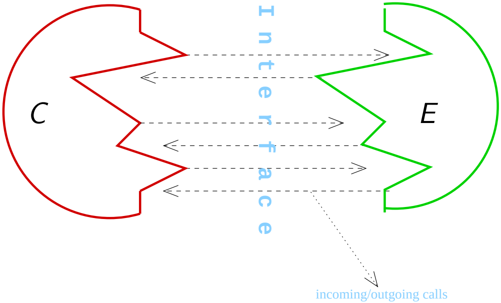
Notion of observation

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

```
class O {        // external  observer
    public void  m(int x) {
      ...
      System.out.println("success");
    }
}
```

Open systems

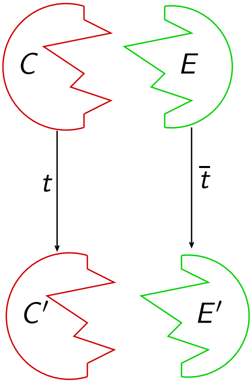
- Component = set of objects + threads "running" in parallel
- Environment = "context" = "observer"



- Component and its environment communicate via *asynchronous method calls*.

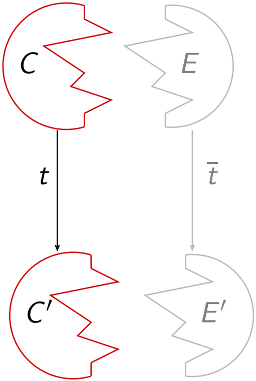
Characterizing the open semantics

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



Characterizing the open semantics

- "message passing"[1] framework ⇒ in first approx.: semantics = message interchange at the interface
- open = environment absent/arbitrary

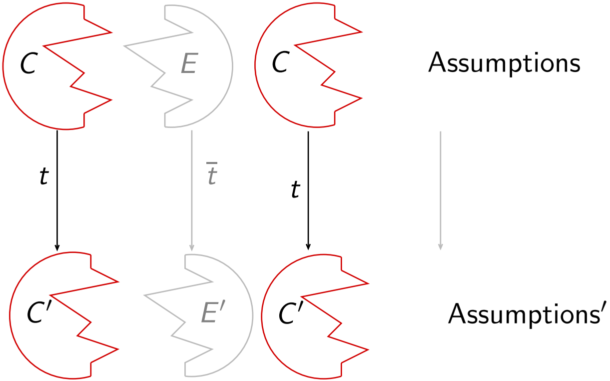


⇒ does this mean: environment behavior arbitrary/chaotic?

[1] no direct access to instance variables

Characterizing the open semantics

- "message passing"[1] framework ⇒ in first approx.: semantics = message interchange at the interface
- open = environment absent/arbitrary



⇒ does this mean: environment behavior arbitrary/chaotic?

[1] no direct access to instance variables

Characterizing the open semantics

- well, depends ...
- does "arbitrary trace" mean $\in \mathit{Label}^*$?
- we know $C \parallel E$ is a program of the language
  - well-formed
  - well-typed
  - class-structured with inheritance
- ultimately: proof of completeness is constructive
  - $\Rightarrow$ formalization of "legal" traces
  - $\Rightarrow$ constructive part: definability: given a trace, program a component that realizes "exactly" this trace.

Open semantics

- operational description:
- assumption/commitment formulation
- $Ass. \vdash C : Comm. \xrightarrow{a} \acute{Ass}. \vdash \acute{C} : \acute{Comm}.$
- interface: 2 orthogonal abstractions:
  - static abstraction: type system
  - dynamic abstraction of heap topology:

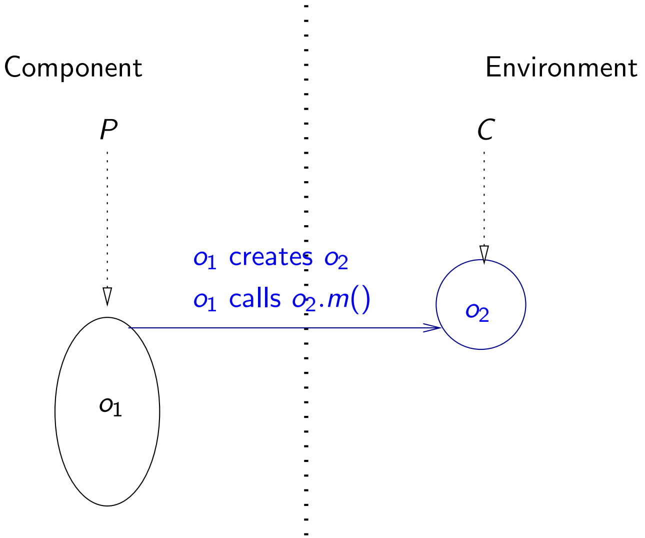
The influence of inheritance

What is the semantical import of classes and inheritance?

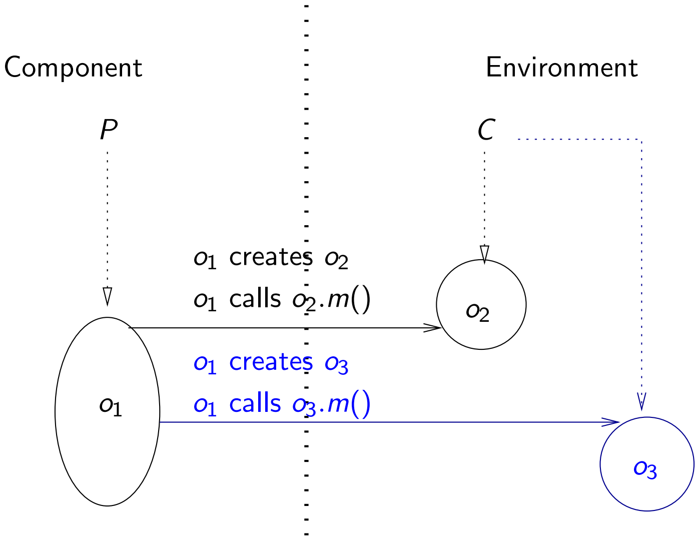
- Interface separates component and observer classes
- Classes are generators of object (via `new`)
- Component classes inherit from environment classes and vice versa.

$\Rightarrow$ instantiation and inheritance as interface interaction

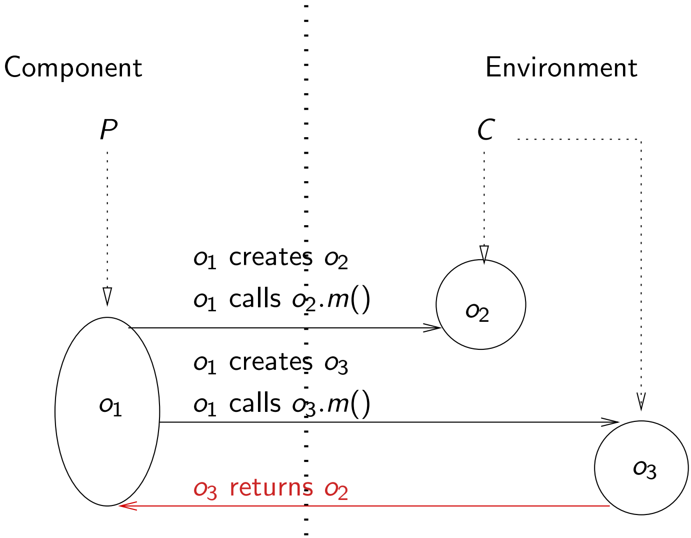
Dynamic heap abstraction example



Dynamic heap abstraction: example

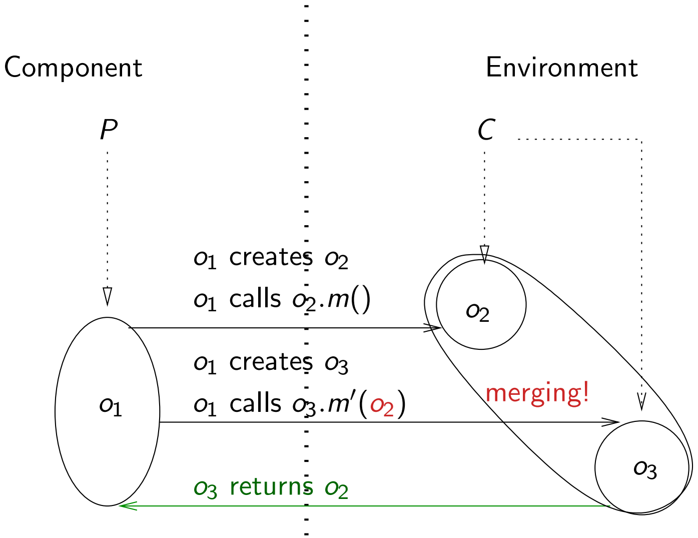


Dynamic heap abstraction: example

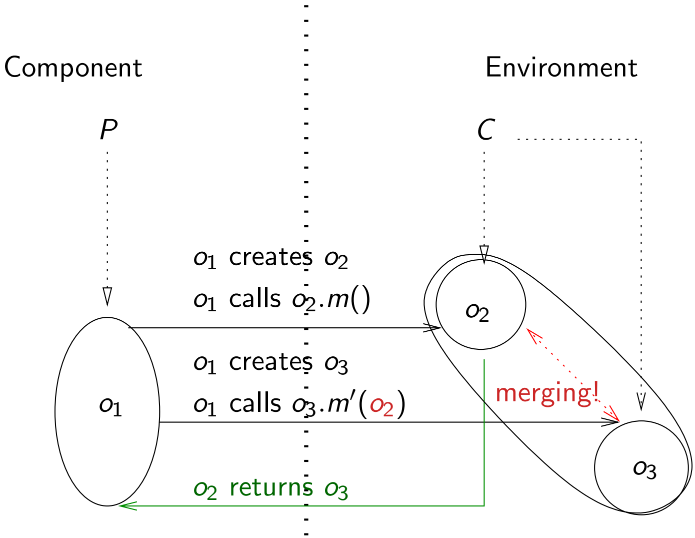


o_2 and o_3 cannot "know" each other!

Dynamic heap abstraction: example



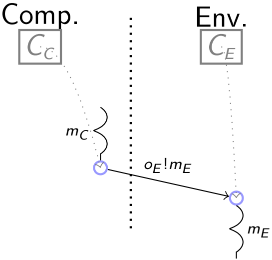
Dynamic heap abstraction: example

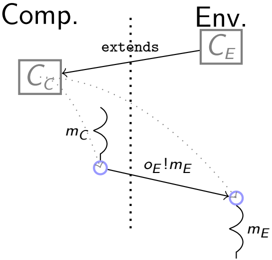


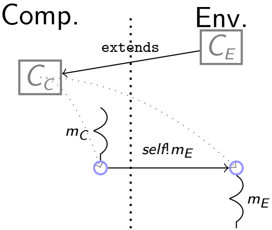
Observability of self-calls

- general intuition: "cross-border" interaction ⇒ interface-interaction
- self-calls: become observable
- cf. also [Viswanathan, 1998]

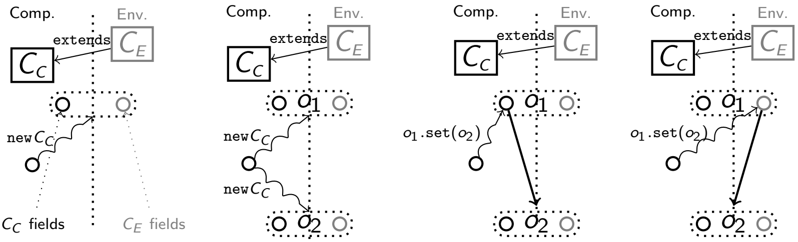
Cross-border inheritance







Cross-border inheritance and heap abstraction



Consequences of inheritance

- separation in component and environment class and cross-border inheritance

  $\Longrightarrow$ self-calls observable.

  $\Longrightarrow$ abstraction of the heap topology

  $\Longrightarrow$ State of an object is split into two halves.

Formal framework: object calculus

- Types and classes:
  - statically typed, only well-typed components are considered
  - classes play role of types and generators of objects
  - single inheritance
- Concurrency: based on active objects/asynchronous method calls
- References:
  - objects and threads have unique names, i.e. identities
  - new objects dynamically allocated on the heap
- Fields are private

Grammar

$$
\begin{array}{rcll}
C & ::= & \mathbf{0} \mid C \parallel C \mid \underline{\nu(n{:}T).}C \mid n[\![O]\!] \mid \underline{n[O, L]} \mid \underline{n\langle t\rangle} & \text{component} \\
O & ::= & n, M, F & \text{object} \\
M & ::= & l = m, \ldots, l = m & \text{method suite} \\
F & ::= & l = f, \ldots, l = f & \text{fields} \\
m & ::= & \varsigma(n{:}T).\lambda(x{:}T, \ldots, x{:}T).t & \text{method} \\
f & ::= & v \mid \bot_{n'} & \text{field} \\
t & ::= & v \mid \text{stop} \mid \text{let } x{:}T = e \text{ in } t & \text{thread} \\
e & ::= & t \mid \text{if } v = v \text{ then } e \text{ else } e \mid \text{if } \mathit{undef}(v.l()) \text{ then } e \text{ else } e & \text{expr.} \\
 & \mid & n@l(\vec{v}) \mid v.l() \mid v.l() := v & \\
 & \mid & \text{new } n \mid \text{claim}@(n, n) \mid \underline{\text{get}@n} \mid \text{suspend}(n) \mid \underline{\text{grab}(n)} \mid \underline{\text{release}(n)} & \\
v & ::= & x \mid n \mid () & \text{values} \\
L & ::= & \bot \mid \top & \text{lock status}
\end{array}
$$

Open semantics and heap abstraction

- Exact interface behavior

$\Rightarrow$ Abstraction of the heap topology necessary

- Keep track of "who has been told what":

$$\Delta; E_\Delta \vdash C : \Theta; E_\Theta$$

- Assumption context: $E_\Delta \subseteq \Delta \times \Delta$ = pairs of objects
- Written $o_1 \hookrightarrow o_2$:
- Worst case: equational theory implied by E_Δ

$$o_1, o_2 \in \Delta : \quad E_\Delta \vdash o_1 \leftrightharpoons o_2$$

Operational semantics and heap abstraction

- as a labeled transition system
- Judgments of the form:

$$\Delta; E_\Delta \vdash C : \Theta; E_\Theta \qquad \text{or short} \qquad \Xi \vdash C$$

Δ and Θ are *name contexts*
E_Δ and E_Θ *connectivity contexts*

External steps

For interaction labels:

γ	$::=$	$p\langle call\ o.l(\vec{v})\rangle \mid p\langle get(v)\rangle \mid \nu(n{:}T)_o$	basic labels
a	$::=$	$\gamma? \mid \gamma!$	receive and send labels

External steps: change of assumption/commitment contexts

- E.g., sending o_1 to o_2, adds $o_2 \hookrightarrow o_1$ to the equations
- outgoing call
  - $a = n\langle call\ o_2.l(o_1)\rangle!$

$$\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta$$

  - assumption update: $\acute{E}_\Delta = E_\Delta + \quad o_2 \hookrightarrow o_1$. We can have definition of assumption update here, similarly for name context check.
- incoming call
  - $a = n\langle call\ o_2.l(o_1)\rangle?$

$$\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta$$

  - assumption check: $E_\Delta \vdash o_2 \hookrightarrow o_1$

External steps: change of assumption/commitment contexts

- E.g., sending o_1 to o_2, adds $o_2 \hookrightarrow o_1$ to the equations
- outgoing call
  - $a = n\langle call\ o_2.l(o_1)\rangle!$

$$\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta$$

  - assumption update: $\acute{E}_\Delta = E_\Delta + \quad o_2 \hookrightarrow o_1$. We can have definition of assumption update here, similarly for name context check.
- incoming call
  - $a = n\langle call\ o_2.l(o_1)\rangle?$

$$\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta$$

  - assumption check: $E_\Delta \vdash o_2 \hookrightarrow o_1$

Some of the external steps

$$\frac{a = p\langle call\ o.l(\vec{v})\rangle? \qquad \Xi \vdash a \qquad \acute{\Xi} = \Xi + a}{\Xi \vdash C \parallel o[c, M, F, \bot] \xrightarrow{a} \acute{\Xi} \vdash C \parallel p\langle \text{let } x{:}T = M.l(o)(\vec{v}) \text{ in release}(o); x\rangle \parallel o[c, M, F,}$$

Simplified rule for incoming call

$$\frac{\begin{array}{c} a = n\langle call\ o_r.l(\vec{v})\rangle? \\ \text{check context:} \quad \Xi \vdash a \\ \text{update contexts:} \quad \acute{\Xi} = \Xi + a \\ \text{semantic step (as in local semantics): from } C \text{ to } \acute{C} \end{array}}{\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash \acute{C}} \textsc{CallI}$$

Some of the external steps

$$\frac{a = p\langle call\ o.l(\vec{v})\rangle? \qquad \Xi \vdash a \qquad \acute{\Xi} = \Xi + a}{\Xi \vdash C \parallel o[c, M, F, \bot] \xrightarrow{a} \acute{\Xi} \vdash C \parallel p\langle \text{let } x{:}T = M.l(o)(\vec{v}) \text{ in release}(o); x\rangle \parallel o[c, M, F,}$$

Simplified rule for incoming call

$$\frac{\begin{array}{c} a = n\langle call\ o_r.l(\vec{v})\rangle? \\ \text{check context: } \quad \Xi \vdash a \\ \text{update contexts: } \quad \acute{\Xi} = \Xi + a \\ \text{semantic step (as in local semantics): from } C \text{ to } \acute{C} \end{array}}{\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash \acute{C}} \textsc{CallI}$$

putting it together: legal traces

- formal system to characterize interface behavior
- judgment:

$$\Xi \vdash a\ s : trace$$

- "after a and with assumption/commitment-contexts Ξ, the trace s is possible"

putting it together: legal traces

$$\Xi \vdash \epsilon : \mathit{trace} \qquad \textsc{L-Empty}$$

$$\frac{a = p\langle \mathit{call}\ o.l(\vec{v})\rangle? \qquad \Xi \vdash a \qquad \acute{\Xi} = \Xi + a \qquad \acute{\Xi} \vdash s : \mathit{trace}}{\Xi \vdash a\ s : \mathit{trace}}\ \textsc{L-CallI}$$

Results

- formalization of open (representation-independent) semantics
  $+$ characterization of possible (legal) interface behavior
- strict separation of assumptions and commitments
- subject reduction
- soundness of abstraction.

References I

[Ábrahám et al., 2008a] Ábrahám, E., Grüner, A., and Steffen, M. (2008a).
Abstract interface behavior of object-oriented languages with monitors.
Theory of Computing Systems, 43(3-4):322–361 (40 pages).

[Ábrahám et al., 2008b] Ábrahám, E., Grüner, A., and Steffen, M. (2008b).
Heap-abstraction for an object-oriented calculus with thread classes.
Journal of Software and Systems Modelling (SoSyM), 7(2):177–208 (32 pages).

[Ábrahám et al., 2011] Ábrahám, E., Mai Thuong Tran, T., and Steffen, M. (2011).
Observable interface behavior and inheritance.
Technical Report 409, University of Oslo, Dept. of Informatics.
www.ifi.uio.no/~msteffen/publications.html#techreports.

[Steffen, 2006] Steffen, M. (2006).
Object-Connectivity and Observability for Class-Based, Object-Oriented Languages.
Habilitation thesis, Technische Faktultät der Christian-Albrechts-Universität zu Kiel.
281 pages.

[Viswanathan, 1998] Viswanathan, R. (1998).
Full abstraction for first-order objects with recursive types and subtyping.
In *Proceedings of LICS '98*. IEEE, Computer Society Press.