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

⇒ Easy question, difficult answer
⇒ 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

⇒ Easy question, difficult answer
⇒ 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

⇒ Easy question, difficult answer
⇒ 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

⇒ Easy question, difficult answer
⇒ Open semantics.
Notion of observation

```java
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");
    }
}
```
- **Component** = set of objects + threads “running” in parallel
- **Environment** = “context” = “observer”

Component and its environment communicate via **asynchronous method calls**.
Corresponding semantics is “traces” as interface interactions (messages, method calls and returns)
"message passing" 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" framework $\Rightarrow$ in first approx.: semantics = message interchange at the interface
- open = environment absent/arbitrary

$C \xrightarrow{t} C'$

$C \xrightarrow{\bar{t}} C'$

$E \xrightarrow{E'}$

Assumptions

$C \xrightarrow{t} C'$

Assumptions'

does this mean: environment behavior arbitrary/chaotic?

$^{1}$no direct access to instance variables
well, depends . . .

does “arbitrary trace” mean $\in Label^*$ ?

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

⇒ instantiation and inheritance as interface interaction
Dynamic heap abstraction example

Component

$P$

$o_1$

$C$

$o_2$

$o_1$ creates $o_2$

$o_1$ calls $o_2.m()$
Dynamic heap abstraction: example

Component

\[ P \]

\[ o_1 \]

\[ o_1 \text{ creates } o_2 \]

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

\[ o_1 \text{ creates } o_3 \]

\[ o_1 \text{ calls } o_3.m() \]

Environment

\[ C \]

\[ o_2 \]

\[ o_3 \]
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 \( o_3.m() \)

Environment

- \( o_2 \)
- \( o_3 \)

- \( o_3 \) returns \( o_2 \)

\( o_2 \) and \( o_3 \) cannot “know” each other!
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 \( o_3.m'(o_2) \)
- \( o_3 \) returns \( o_2 \)

Environment

\[ C \]

- \( o_2 \)
- \( o_3 \)

merging!
**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

- $C$
- $o_2$
- $o_3$

Merging!
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

\[
\text{Comp. } C_C \text{ extends } C_E \\
\text{new } C_C \rightarrow \text{fields}
\]

\[
\text{Env. } C_E \text{ fields}
\]

\[
\text{Comp. } C_C \text{ extends } C_E \\
o_1 \rightarrow \text{fields}
\]

\[
o_2 \rightarrow \text{fields}
\]

\[
o_1.\text{set}(o_2)
\]

\[
\text{Comp. } C_C \text{ extends } C_E \\
o_1 \rightarrow \text{fields}
\]

\[
o_2 \rightarrow \text{fields}
\]

\[
o_1.\text{set}(o_2)
\]
Consequences of inheritance

- separation in component and environment class and cross-border inheritance
  - self-calls observable.
  - abstraction of the heap topology
  - 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

\[
C ::= \ 0 \ | \ C \ || \ C \ | \ \nu(n:T).C \ | \ n(O) \ | \ n(O,L) \ | \ n(t) \ \\
O ::= \ n, M, F \ \\
M ::= \ l = m, \ldots, l = m \ \\
F ::= \ l = f, \ldots, l = f \ \\
m ::= \ \varsigma(n:T).\lambda(x:T, \ldots, x:T).t \ \\
f ::= \ v \ | \ \bot' \ \\
t ::= \ v \ | \ \text{stop} \ | \ \text{let} \ x : T = e \ \text{in} \ t \ \\
e ::= \ t \ | \ \text{if} \ v = v \ \text{then} \ e \ \text{else} \ e \ | \ \text{if} \ \text{undef}(v.l()) \ \text{then} \ e \ \text{else} \ e \ \\
| \ n@l(\bar{v}) \ | \ v.l() \ | \ v.l() := v \ \\
| \ \text{new} \ n \ | \ \text{claim}@\!(n, n) \ | \ \text{get}@n \ | \ \text{suspend}(n) \ | \ \text{grab}(n) \ | \ \text{release}(n) \ \\
v ::= \ x \ | \ n \ | \ () \ \\
L ::= \ \bot \ | \ T \ 
\]
Open semantics and heap abstraction

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

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

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

\[ o_1, o_2 \in \Delta : \quad E_\Delta \vdash o_1 \leftrightarrow o_2 \]
as a labeled transition system

Judgments of the form:

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

or short

\[ \Xi \vdash C \]

\( \Delta \) and \( \Theta \) are name contexts

\( E_\Delta \) and \( E_\Theta \) connectivity contexts
External steps

For interaction labels:

\[ \gamma ::= p\langle \text{call } o.l(\vec{v}) \rangle \mid p\langle \text{get}(v) \rangle \mid \nu(n:T)_o \]

\[ a ::= \gamma? \mid \gamma! \]
External steps: change of assumption/commitment contexts

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

$$\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{a} \Delta'; E_\Delta \vdash C : \Theta; E_\Theta$$

- assumption update: $E_\Delta = E_\Delta + o_2 \leftrightarrow o_1$. We can have definition of assumption update here, similarly for name context check.

- incoming call
  - $a = n\langle \text{call } o_2.l(o_1) \rangle?$

$$\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{a} \Delta'; E_\Delta \vdash C : \Theta; E_\Theta$$

- assumption check: $E_\Delta \vdash o_2 \leftrightarrow o_1$
External steps: change of assumption/commitment contexts

- E.g., sending $o_1$ to $o_2$, adds $o_2 \leftrightarrow o_1$ to the equations

  **outgoing call**
  
  $a = n\langle call \ o_2.l(o_1)\rangle !$

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

  **assumption update:** $E'_\Delta = E_\Delta + o_2 \leftrightarrow o_1$. We can have definition of assumption update here, similarly for name context check.

  **incoming call**

  $a = n\langle call \ o_2.l(o_1)\rangle ?$

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

  **assumption check:** $E_\Delta \vdash o_2 \leftrightarrow o_1$
Some of the external steps

\[ a = p\langle \text{call } o.l(\vec{v})\rangle? \quad \Xi \vdash a \quad \dot{\Xi} = \Xi + a \]

\[ \Xi \vdash C \parallel o[c, M, F, \perp] \xrightarrow{a} \dot{\Xi} \vdash C \parallel p\langle \text{let } x : T = M.l(o)(\vec{v}) \text{ in release}(o); x \rangle \parallel o[c, M, F, \top] \]

Simplified rule for incoming call

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

check context: \[ \Xi \vdash a \]

update contexts: \[ \dot{\Xi} = \Xi + a \]

semantic step (as in local semantics): from \( C \) to \( \dot{C} \)

\[ \Xi \vdash C \xrightarrow{a} \dot{\Xi} \vdash \dot{C} \]
Some of the external steps

\[
\begin{align*}
  a &= p\langle \text{call } o.l(\vec{v}) \rangle? \\
  \Xi \vdash a & \quad \dot{\Xi} = \Xi + a
\end{align*}
\]

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

Simplified rule for incoming call

\[
\begin{align*}
  a &= n\langle \text{call } o_r.l(\vec{v}) \rangle? \\
  \text{check context: } & \quad \Xi \vdash a \\
  \text{update contexts: } & \quad \dot{\Xi} = \Xi + a \\
  \text{semantic step (as in local semantics): from } C \text{ to } \dot{C} & \quad \text{CALLI}
\end{align*}
\]

\[\Xi \vdash C \xrightarrow{a} \Xi \vdash \dot{C}\]
formal system to characterize interface behavior

judgment:

\[ \Xi \vdash a \ s : \text{trace} \]

“after a and with assumption/commitment-contexts \( \Xi \), the trace \( s \) is possible”
\[\Gamma \vdash \epsilon : trace \quad \text{L-EMPTY}\]

\[a = p\langle \text{call o.I} (\vec{v}) \rangle ? \quad \Gamma \vdash a \quad \check{\Xi} = \check{\Xi} + a \quad \check{\Xi} \vdash s : trace \quad \text{L-CALLI}\]

\[\Gamma \vdash a \ s : trace\]
Results

- formalization of open (representation-independent) semantics
  + characterization of possible (legal) interface behavior
- strict separation of assumptions and commitments
- subject reduction
- soundness of abstraction.
Abstract interface behavior of object-oriented languages with monitors.

Heap-abstraction for an object-oriented calculus with thread classes.

Observable interface behavior and inheritance.
Technical Report 409, University of Oslo, Dept. of Informatics.
www.ifi.uio.no/~msteffen/publications.html#techreports.

Object-Connectivity and Observability for Class-Based, Object-Oriented Languages.
281 pages.

Full abstraction for first-order objects with recursive types and subtyping.