Observability and Full-Abstraction for Object-Oriented Languages

Martin Steffen

University of Oslo, Norway

College of Technology, Vietnam National University, Hà-nội, Việt Nam

October 2010
Observability

What is observable in an (oo) language?

- easy question, difficult answer
- compositionality, replacement
- full abstraction
- proof theory, completeness, realizability
Language features

- sequential programs
- concurrency
- objects
- classes
- locks/monitors
- cloning
Language features

<table>
<thead>
<tr>
<th>sequential programs</th>
<th>⇒</th>
<th>state-transformers, continuous functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>concurrency</td>
<td>⇒</td>
<td></td>
</tr>
<tr>
<td>objects</td>
<td>⇒</td>
<td></td>
</tr>
<tr>
<td>classes</td>
<td>⇒</td>
<td></td>
</tr>
<tr>
<td>locks/monitors</td>
<td>⇒</td>
<td></td>
</tr>
<tr>
<td>cloning</td>
<td>⇒</td>
<td></td>
</tr>
</tbody>
</table>
Language features

| sequential programs | ⇒ | state-transformers, continuous functions |
| concurrency         | ⇒ | “traces” |
| objects             | ⇒ |
| classes             | ⇒ |
| locks/monitors      | ⇒ |
| cloning             | ⇒ |
## Language features

| sequential programs | ⇒ | state-transformers, continuous functions |
| concurrency | ⇒ | “traces” |
| objects | ⇒ | “traces” |
| classes | ⇒ | |
| locks/monitors | ⇒ | |
| cloning | ⇒ | |
## Language features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential programs</td>
<td>state-transformers, continuous functions</td>
</tr>
<tr>
<td>Concurrency</td>
<td>“traces”</td>
</tr>
<tr>
<td>Objects</td>
<td>“traces”</td>
</tr>
<tr>
<td>Classes</td>
<td>“connectivity”, abstract heap</td>
</tr>
<tr>
<td>Locks/monitors</td>
<td></td>
</tr>
<tr>
<td>Cloning</td>
<td></td>
</tr>
<tr>
<td>Sequential programs ⇒ state-transformers, continuous functions</td>
<td></td>
</tr>
<tr>
<td>Concurrency ⇒ &quot;traces&quot;</td>
<td></td>
</tr>
<tr>
<td>Objects ⇒ &quot;traces&quot;</td>
<td></td>
</tr>
<tr>
<td>Classes ⇒ &quot;connectivity&quot;, abstract heap</td>
<td></td>
</tr>
<tr>
<td>Locks/monitors ⇒ tricky dependencies to capture mutex</td>
<td></td>
</tr>
<tr>
<td>Cloning ⇒</td>
<td></td>
</tr>
</tbody>
</table>
Language features

- **Sequential programs** ⇒ state-transformers, continuous functions
- **Concurrency** ⇒ “traces”
- **Objects** ⇒ “traces”
- **Classes** ⇒ “connectivity”, abstract
- **Locks/monitors** ⇒ tricky dependencies to capture
- **Cloning** ⇒ [“branching”]
- **Heap**
Overview

1 Introduction

2 Facets of object-orientation
   - Classes
   - Concurrency and locks
   - Data dependencies
   - Control dependencies
   - Inheritance

3 Conclusion
public class P {  // 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) {
        <some code>;  // body of m
        System.out.println("success");
    }
}
Notion of observation

- pretty simple observational notion: “may-testing”:
  - compose a program with a context/observer, let both run and see, whether the observer may be/is successful

- $P_1 \subseteq_{\text{may}} P_2$: for all observers $O$: if $P_1 + O$ may be/is successful, then so may be/is $P_2 + O$.

- observational
  - “black-box”
  - fundamental distinction between program/component/player vs. environment/context/observer/opponent
3 problems to tackle for an open f-a 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
3 problems to tackle for an open f-a 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?
- well, depends . . .

\(^1\)no direct access to instance variables
3 problems to tackle for an open f-a semantics

- "message passing"\(^1\) framework \(\Rightarrow\) in first approx.: semantics = message interchange at the interface

- open = environment absent / arbitrary

\(\Rightarrow\) does this mean: environment behavior arbitrary/chaotic?

- does “arbitrary trace” mean \(\in Label^*\) ?

\(^1\) no direct access to instance variables
3 problems to tackle for an open f-a 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?
- we know \(P + O\) is a program of the language
  - well-formed
  - well-typed
  - class-structured
- ultimately: proof of completeness is constructive
  - ⇒ formalization of “legal” traces
  - ⇒ uncertainty of observation/closure conditions
  - ⇒ constructive part: definability: given a trace, program a component that realizes “exactly” this trace.

\(^1\)no direct access to instance variables
Open semantics

- operational description:
- assumption/commitment formulation

\[ \text{Ass.} \vdash C : \text{Comm.} \xrightarrow{a} \text{Ass.} \vdash \acute{C} : \text{Comm.} \]

interface: 3 orthogonal abstractions:

- static abstraction: type system
- dynamic abstraction of heap topology:
- abstraction of the stack structure of thread(s): enabledness conditions ("well-bracketed condition")
Formal framework: object calculus

- **Origin:**
  - Abadi/Cardelli: simple imperative object calculus \( \text{imp} \)
  - Gordon/Hankin: concurrent variant \( \text{conc} \)
  - Jeffrey/Rathke: fully abstract may testing semantics
  - Here: extended with classes and thread classes

- **Types and classes:**
  - Statically typed, only well-typed components are considered
  - Classes play role of types and generators of objects

- **Concurrency:**
  - Based on threads
  - Spawned by means of thread classes

- **References:**
  - Objects and threads have unique names, i.e. identities
  - New objects dynamically allocated on the heap

- **Fields are private**
Grammar

\[ C ::= 0 | C \parallel C | \nu(n:T).C | n[\langle O \rangle] | n[n, F] | n\langle t \rangle \] program

\[ O ::= F, M \] object

\[ M ::= l = m, \ldots, l = m \] method suite

\[ F ::= l = f, \ldots, l = f \] fields

\[ m ::= \varsigma(n:T).\lambda(x:T, \ldots, x:T).t \] method

\[ f ::= \varsigma(n:T).\lambda().\perp_c | fv \] field

\[ fv ::= \varsigma(n:T).\lambda().v \] defined field

\[ t ::= v | \text{stop} | \text{let } x: T = e \text{ in } t \] thread

\[ e ::= t | \text{if } v = v \text{ then } e \text{ else } e | \text{if undef}(v.l) \text{ then } e \text{ else } e \] expression

\[ | v.l(v, \ldots, v) | v.l := fv \] currentthread

\[ | \text{new } n | \text{new}\langle t \rangle \]

\[ v ::= x | n \] values
open semantics (based onij may testing): in principle: easy and understood
⇒ corresponding semantics is “traces” as interface interactions (messages, method calls and returns)

what is the semantical import of classes?

3 issues:
1. interface separates observer and component classes
⇒ instantiation requests as interface interaction
2. class = generators of object (via new)² ⇒ replay
3. abstraction of the heap topology

²Classes in Java or C# serve also as kind of types, and furthermore for inheritance. We ignore that mostly here.
Cross-border instantiation: Java-Example

Component:

class C {
    public void m() {
        Obs o = new Obs();
        ...
    }
}

Observer:

public class Obs {
    public static void main(String[] arg) {
        C c = new C();
        c.m();
    }
}
Cross-border instantiation & heap abstraction

- **classes** as unit of code/exchange
- **instantiation** as interface interaction
- **component** instantiates **observer class** ⇒
  - **instance**: part of the **observer**
  - **reference** to it: kept at the **component**
Cross-border instantiation & heap abstraction

Program

Environment

new

$c_2$
Heap separation

- heap is separated in component and environment part:

- comp. resp. env. objects can be grouped in "cliques"
- restriction on possible component-environment interactions
Where are we?

- open semantics in the presence of classes ⇒ abstraction of heap topology

- features (Java/C#-inspired):
  - objects and classes
  - (single/multiple) threads
  - references/heap/aliasing
  - typed language

- formalized in some “object calculus”
Order of events

- separate observer cliques
- separate observer cliques cannot cooperate

⇒ order of interaction not globally observable

\[ \Theta \quad \Delta \]

\[ c_1 \quad c_2 \quad \Theta \]

\[ \Theta \quad \Delta \]

\[ c_1 \quad c_2 \quad \Theta \]

\[ \Theta \quad \Delta \]

**Note:** Take care of merging
Order of events

- separate observer cliques
- separate observer cliques cannot cooperate

$\Rightarrow$ order of interaction not globally observable\(^3\)

\(^3\)Take care of merging
Order of events

- separate observer cliques
- separate observer cliques cannot cooperate

$\Rightarrow$ order of interaction not globally observable$^3$

$\Theta$ $\Delta$

$\Theta$ $\Delta$

$\Theta$ $\Delta$

$^3$Take care of merging
Order of events

- separate observer cliques
- separate observer cliques cannot cooperate

$\Rightarrow$ order of interaction not globally observable

$\Theta$ $\Delta$

$\Theta$ $\Delta$

$\Theta$ $\Delta$

$^3$Take care of merging
Classes as generators of objects

- two new instances of a class are identical up-to their id
- for the observer:

  what can be observed once by one observer clique,
  can be observed again (up-to identity) by a second
  “instance” of the observer
two new instances of a class are identical up-to their id

for the observer:

what can be observed once by one observer clique, can be observed again (up-to identity) by a second “instance” of the observer
two new instances of a class are identical up-to their id

for the observer:

what can be observed once by one observer clique, can be observed again (up-to identity) by a second “instance” of the observer
Swapping and replay

\[ \Xi_0 \vdash \Theta \ s \triangleright t_1 \not\equiv t_2 \quad \vdash u : wbalanced \quad \vdash t_1, t_2 : p wbalanced^p \]

\[ \Xi_0 \vdash s \ \nu(\Phi).t_1 \ t_2 \ u \ \triangleleft \Theta \ s \ \nu(\Phi).t_2 \ t_1 \ u \]

\[ \Xi_0 \vdash \Theta \ s \triangleright t_1 \not\equiv t_2 \quad \vdash t_1 : balanced \quad |t_2|\ even \]

\[ \Xi_0 \vdash s \ \nu(\Phi).t_1 \ t_2 \ u \ \triangleleft \Theta \ s \nu(\Phi).t_2 \ t_1 \ u \]

\[ sender(s\gamma?) = o \quad s \downarrow_\sigma \gamma? \not\triangleleft s \downarrow_\sigma' \]

\[ \Xi \vdash s\gamma? \not\triangleleft_\Delta s \]

\[ \Xi \vdash s\gamma? \not\triangleleft_\Delta s \]

\[ \Xi_0 \vdash \Theta s \nu(\Phi).t_1 \ t_2 \ u \ \triangleleft \Theta s \nu(\Phi).t_2 \ t_1 \ u \]
Single-threaded case

- Programs become deterministic
- Behavior becomes (somewhat) simpler, more restricted
- Problematic:
  - Characterization of allowed interface behavior ("legal traces")
  - Definability: single-threaded programming of the observer
Deterministic traces

- more than one instance of a class in one trace
- different instances of same class behave “the same”
- equivalent stimulus/input history ⇒ 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

- more than one instance of a class in one trace
- different instances of same class behave “the same”
- equivalent stimulus/input history ⇒ equivalent reaction
- example:

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


Deterministic traces

- 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:

$$\nu(o:c)n\langle \text{call } o.l() \rangle? \nu(o_2:c_2)n\langle \text{return}(o_2) \rangle! \quad \nu(o':c)l\langle \text{call } o'.l() \rangle? \nu(o'_2:c_2)l\langle \text{return}(o'_2) \rangle!$$
Deterministic traces

- 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:

  - deterministic reaction **per clique** of objects
  $\Rightarrow$ history of clique only up-to swapping:
Given label $a = \gamma$ and trace $ra$ with $\Delta \vdash r \triangleright a : \Theta$. Trace $r$ can be extended deterministically by $a$, written $\Delta \vdash r \triangleright a : \text{det}_\Delta \Theta$, if the following holds:

$$\Delta; E_\Delta \vdash ra \cong_\Delta r : \Theta; E_\Theta \quad \text{or} \quad \neg \exists \text{ label } b \text{ with } \Delta; E_\Delta \vdash rb \cong_\Delta r : \Theta; E_\Theta$$

(1)

(dual for outgoing communications $a$, especially: $\cong_\Theta$ instead of $\cong_\Delta$)
Monitors

- shared (instance) state + concurrency ⇒ mutex
- sync. mechanism: monitors
  - for instance in Java
  - here
    - no synchronized blocks
    - no wait / signal
    - no connectivity
- but:
  - re-entrant monitors (recursion)

---

\(^4\)In Java: wait and notify.
What changes?

Now:

Does the addition of monitors increase or decrease the discriminating power or not?

Intuitively: 2 plausible answers:
- the observer sees less!
- the observer sees more!

Different angle: it does not matter
Now:

Does the addition of monitors increase or decrease the discriminating power or not?

intuitively: 2 plausible answers:

- the observer sees less!
- the observer sees more!

different angle: it does not matter
What changes?

Now:

*Does the addition of monitors* increase or decrease the discriminating power or not?

Intuitively: 2 plausible answers:

- the observer sees less!
- the observer sees more!

Different angle: it does not matter
What changes?

Now:

Does the addition of monitors increase or decrease the discriminating power or not?

intuitively: 2 plausible answers:
- the observer sees less!
- the observer sees more!

different angle: it does not matter
incorporate *monitors* into the semantics characterization of the interface behavior

- **may** and **must** approximation of *lock-ownership*

**design goals**

- (preferably) **seamless** extension of the **calculus** with an eye to **compositionality**

⇒ clean separation of concerns between

*assumptions vs. commitments*

intuitively:

- enabledness of *input* must depend **only** on the *environment* (= assumption)
- enabledness of *output* must depend **only** on the *component* (= commitments)

interface *trace* must contain all relevant information relevant (and **not** part of the internal state(s))
Example (1)

- 2 calls, competing for the same (component) lock
- data dependence
  - $o'$ received by the first call (of $n_1$)
  - returned by second thread $n_2$ afterwards
- note: $o'$ is new

```
program environment
\gamma_{c_1}^? \quad \gamma_{c_2}^? \quad \gamma'_{c_1} \quad \gamma_{r_2}!
```

- question: is that trace possible?
Example (1)

- 2 calls, competing for the same (component) lock
- **data dependence**
  - $o'$ received by the first call (of $n_1$)
  - returned by second thread $n_2$ afterwards
- **note:** $o'$ is **new**

$$
\gamma_{c_1} ? \gamma_{c_2} ? \gamma'_{c_1} ! \gamma_{n_2} ! = (\nu o' : c) n_1 \langle \text{call } o.l(o') \rangle ? n_2 \langle \text{call } o.l() \rangle ? n_1 \langle \text{call } \tilde{o}.l() \rangle ! n_2 \langle \text{return}(o') \rangle !
$$

- **question:** *is that trace possible?*
$\gamma_{c_1} \gamma_{c_2} \gamma'_{c_1} \gamma_{r_2} = (\nu o' : c) n_1 \langle \text{call } o.l(o') \rangle? n_2 \langle \text{call } o.l() \rangle? n_1 \langle \text{call } o.l() \rangle! n_2 \langle \text{return}(o') \rangle!$

- **question:** *is that trace possible?*
- **the answer is** *no!*
- **data:** “$n_1 \text{ before } n_2$”
- **monitors:**
  - the outgoing call of $n_1$ shows that $n_1$ must have the lock now
  - $\Rightarrow$ $n_2$ cannot have it now: $\Rightarrow$
    - “$n_2 \text{ before } n_1$”
Example (1)

\[
\begin{align*}
\gamma_{c_1} & \ ? \gamma_{c_2} \ ? \ \gamma'_{c_1} \ ! \ \gamma_{r_2} ! = \\
(\nu o' : c) n_1 \langle \text{call } o.l(o') \rangle & \ ? \ n_2 \langle \text{call } o.l() \rangle \ ? \ n_1 \langle \text{call } \tilde{o}.l() \rangle ! \ n_2 \langle \text{return}(o') \rangle ! 
\end{align*}
\]

- question: *is that trace possible?*

\[
\begin{align*}
\gamma_{c_1} ? & \quad \gamma_{c_2} ?
\end{align*}
\]

(2)

**Note:** *non-atomic lock-grabbing \(\Rightarrow\) no order!*
Example (1)

\[
\gamma_{c_1} \gamma_{c_2} \gamma'_{c_1} \gamma_{r_2} = (\nu o' : c) n_1 \langle \text{call } o.l(o') \rangle \otimes n_2 \langle \text{call } o.l() \rangle \otimes n_1 \langle \text{call } \bar{o}.l() \rangle \otimes n_2 \langle \text{return}(o') \rangle
\]

question: is that trace possible?

Note: there is no order between events of \(n_1\) and \(n_2\)!
Example (1)

\[ \gamma_1 \gamma_2 ? \gamma_1' ! \gamma_2' ! = (\nu o':c)n_1 \langle \text{call } o.l(o') \rangle? n_2 \langle \text{call } o.l() \rangle? n_1 \langle \text{call } \tilde{o}.l() \rangle! n_2 \langle \text{return}(o') \rangle! \]

question: is that trace possible?

(4)

Note:
- data dependence because of \( o' \)
Conditions characterizing monitors

- apart from conditions concerning non-monitor features
  - well-typedness
  - freshness
  - (connectivity)

- 3 types of dependencies / precedences between events

1. **mutual exclusion:**
   *If a thread has taken the lock of a monitor, interactions of other threads with that monitor must either occur **before** the lock is taken, or **after** it has been released again.*

2. **data dependencies:**
   *No value (unless generated new or being “constant/global”) can be transmitted **before** it has been received.*

3. **control dependencies:** (program order)
   *Within 1 thread, the events are linearly ordered.*
Lock ownership

- **question:**
  
  *given interaction of thread n, is the lock of object o available*

- first attempt:
  
  *"after call n⟨call o.l()⟩?, thread n owns the lock of o."*

- alas: not true!

- **complication:** non-atomic lock-grabbing

- **handing-over call ⇒ not necessarily obtaining lock**
Lock ownership: non-atomic lock grabbing

- **delayed** observation:
  
  \[
  n\langle \text{call } o.l() \rangle ?
  \]
  
  "after \[ n\langle \text{call } o.l() \rangle ? \], thread \( n \) \textit{may} own lock of component object \( o \)."

- and **later**:
  
  \[
  n\langle \text{call } o.l() \rangle ? n\langle \text{call } o'.l() \rangle !
  \]
  
  "after \[ n\langle \text{call } o.l() \rangle ? n\langle \text{call } o'.l() \rangle ! \], thread \( n \) \textit{must} own lock of \( o \).

2 approximations per thread:

- **potential** lock-ownership: "\textit{may}"”, written: \( \diamondsuit_{nO} \)

- **necessary** lock-ownership: "\textit{must}", written: \( \Box_{nO} \)
given the trace $t$ projected to one thread from the component-perspective\(^5\)

after $s$, the thread may own the lock of $o$:

\[
\Sigma \vdash s : \Diamond o
\]

\(^5\)dually for the environment.
Lock-ownership: May-approximation

\[
\begin{align*}
\therefore & \quad s_2 : balanced \quad s_2 \neq \epsilon \quad \triangleright s_1 : \Diamond o \\
\therefore & \quad s_1 \quad s_2 : \Diamond o \quad \text{M-\Diamond} \\
\therefore & \quad \text{receiver}(s \gamma_c?) = o \quad \therefore \quad s \gamma_c? : \Diamond o \quad \text{M-I\Diamond}_1 \\
\therefore & \quad \text{receiver}(s \gamma_c?) \neq o \quad \triangleright s : \Diamond o \\
\therefore & \quad \triangleright s \gamma_c? : \Diamond o \quad \text{M-I\Diamond}_2 \\
\therefore & \quad \triangleright s : \Diamond o \quad \text{M-O\Diamond} \\
\therefore & \quad \triangleright s \gamma_c! : \Diamond o
\end{align*}
\]
similar system as in the may case
based on the may-system\(^5\)
again from the component-perspective
after \(s\), the thread must own the lock of \(o\):

\[\vdash s : \Box_o\]

\(^5\)but no mutual recursion
Lock-ownership: Must-approximation

\[\vdash t : \square o\]
\[\vdash t \gamma_c ? : \square o\]  \text{M-I} \square_1

\[\vdash t : \diamond o\]
\[\vdash t \gamma_c ! : \square o\]  \text{M-O} \square_1

\[\vdash t \gamma_r ? : \square o\]  \text{M-I} \square_2

\[\vdash t \gamma_r ? : \square o\]  \text{M-O} \square_2

\[\vdash t \gamma_r ? : \square o\]  \text{M-O} \square_2
**Illustration**

**Example**

\[ t = \gamma_c? = (\nu \Xi)n\langle \text{call o_r.l(o)}\rangle? . \]

then

\[ \Xi \vdash t : \diamond_o \quad \text{and} \quad \Xi \vdash t : \neg \diamond o \]

Note: \( \diamond \) is a *local* interpretation.

---

**Example**

\[ t = \gamma_c?\gamma_r! = (\nu \Xi)n\langle \text{call o_r.l}()\rangle? n\langle \text{return}()\rangle! . \]

Then:

\[ \Xi \vdash \gamma_c? : \diamond_n o_r \quad \text{but} \quad \Xi \not\vdash \gamma_c? : \Box_n o_r \]

and

\[ \Xi \vdash t : \neg \diamond_o r \]
Mutual exclusion

- here: again for **component** locks
- "global" perspective: not just one thread
- *mutex* precedence edges for event *a* after *r* wrt. component object *o*.

\[ M_\Theta(ra, o) \]

- auxiliary definitions:
  - "after may": \( \Diamond(t, o) \)
  - "before must": \( \Box(t, o) \)

- edges: \( \vdash a_1 \rightarrow^m a_2 \)

- distinction for *a* between incoming communication
  - no condition for incoming returns
  - incoming calls

- outgoing communication: 2 conditions
  - *a* before other threads have taken the lock
  - after
Mutual exclusion

\[ M_\Theta(r\gamma_c?, o) = \diamondsuit \neq n(r, o) \rightarrow \gamma_c? \]
\[ M_\Theta(r\gamma_r?, o) = \{\} \]

\[ M_\Theta(r\gamma!, o) = \gamma! \rightarrow \lozenge \neq n(r, o), \]
\[ \diamondsuit \neq n(r, o) \rightarrow \square n(r\gamma!, o) \]
data dependence

- jugment

\[ \vdash_\Theta r : \gamma ? \rightarrow^d o \]

if \( o \in \text{names}(\gamma) \) and \( r' \gamma ? \) is a prefix of \( r \).
- “\( o \) is potentially data-dependent on event/label \( \gamma ? \) of trace \( r \)”
- note: it’s only potential dependence

\[
D_\Theta(r \gamma !) = \{ \bar{\gamma} ? \rightarrow \gamma ! \} \quad \text{where} \quad \vdash_\Theta \bar{\gamma} ? \rightarrow^d \text{fn}(\gamma !) \cap \Delta(r) \\
D_\Theta(r \gamma ?) = \{ \} .
\]

For \( \Delta \), the definitions are applied dually.
control dependencies

- precedence nr. 3
- trivial

⇒ the events within each trace are \textit{linearly} ordered

- notation

\[
\vdash a' \rightarrow^c a
\]
putting it together: legal traces

- formal system to characterize interface behavior
- non-branching :-)
- judgment:
  \[ \Theta; G \vdash r \triangleright s : \text{trace} \]
- “after \( r \) and with assumption / commitment -contexts \( \Theta \) and \( G \), the trace \( s \) is possible”
- context \( G \):
  - precedence graphs
  - cleanly separated into \( G_\Delta \) and \( G_\Theta \)
  - 3 reasons for precedence:
    1. \( \rightarrow^m \)
    2. \( \rightarrow^d \)
    3. \( \rightarrow^c \)
- \( G \) must remain acyclic: \( \vdash G \text{ ok} \)
\[
\Gamma \vdash r \triangleright o_s \xrightarrow{a} o_r \quad \delta = \delta' + a \quad \Gamma \vdash a : \text{wt}
\]
\[
\hat{\Gamma}_\Theta = \Gamma_\Theta \cup \Gamma_\Theta(ra, o_r) \quad \hat{\Gamma}_\Delta = \Gamma_\Delta \cup \Gamma_\Delta(ra, o_s) \quad \Gamma \vdash \hat{\Gamma}_\Delta : \text{wt}
\]
\[
a = \nu(\delta'). \ n\langle \text{call } o_r. l(\vec{v})\rangle? \quad \Gamma \vdash r a \triangleright s : \text{trace}
\]
\[
\Gamma ; G \vdash r \triangleright a s : \text{trace}
\]

L-CALLI
Inheritance

- core oo mechanism
- code reuse
- sometimes mixed-up with sub-typing
- various flavors
- not undisputed
Fragile base class problem

class A {
    void add () {...}
    void add2 () {...}
    ...
}
class B extends A {
    void add () {
        size = size + 1;
        super.add();
    }
    void add2 () {
        size = size + 2;
        super.add2();
    }
}
Bottom line

“inheritance breaks encapsulation”

- life made easier by: *no re-entrance*
- “private” vs. public fields
- private vs. public methods
- `super`-keyword
- “shadowing”: binding of methods vs. binding of field
Language

- active objects language
- Creol-“dialect”
- async. methods, futures
- no interfaces (at user level)
- “private” fields, private and public methods
- no super
Fields and shadowing

class C1 {
    x;
    m() {... x...}
}

class C2 extends C1 {
    x;  // overriding/shadowing
    n() {... m() ...}
}
class $C_1$ {
  x;
  getx() { x }
  m() { .. self.getx() ... }
}

class $C_2$ extends $C_1$ {
  x;
  getx() { x }
  n() { ... m() ... }
}
class C1 {
    String s = "C1";
    private void n() { System.out.print("C1");}
    void m() { this.n();}
}

class C2 extends C1 {
    void n() { System.out.print("C2");}
}
Subtype polymorphism

Assume $C_2 \leq C_1$. Can one observe the run-time type?

$$\text{let } x : C_1 = \text{new } C_1 \quad \text{vs.} \quad \text{let } x : C_1 = \text{new } C_2$$ (2)
public class DynamicTypeObs1 {
    public static void main(String[] args) {
        C1 c = new C2();
        c.m();
    }
}

class C1 {
    void m() { System.out.print("C1"); }
}

class C2 extends C1 {
    void m() { System.out.print("C2"); }
}
Observability of self-calls

- general intuition: “cross-border” interaction $\Rightarrow$ interface-interaction
- self-calls: not observable
- influence of super
- cf. also [Viswanathan, 1998]

```java
class C1 {
    ..
    public void m1 () { ... self.m2 ... }
    public void m2 () { ... }
}

class C2 extends C1 {
    ..
    public void m2 () { .... }
    ..
}
```
Cross-border inheritance

$\Theta \Delta$

$\Theta \quad \Delta$

$\Theta$

$\Theta$ $\rightarrow$ $\Delta$

$\Theta$ $\rightarrow$ $o_2$

$o_2 \rightarrow o_3'$

$o_2'$ $\rightarrow$ $o_3'$
“Splittin” of object states
Consequences of inheritance

- self-calls observable
- run-time types and class hierarchy (partially) observable
- connectivity of objects
- what exactly is observable depends a lot on the details.
Results

- observability & full abstraction for various aspects
  - classes
  - multi-threading vs. sequential oo-programs
  - thread-classes
  - monitors
  - active objects, futures, and promises
  - inheritance (under work)
What is it all good for?

- “reference-point” semantics
- guideline for “good language design”
- compositional/modular (Hoare) proof systems
  [Ábrahám et al., 2005]
- Testing [Grüner, 2010] [Torjusen, 2010]
related work

- vast body of full abstraction results
- for OO: [Jeffrey and Rathke, 2002] [Jeffrey and Rathke, 2005]
- inheritance [Viswanathan, 1998] [Balzarotti et al., 1999]
- Game semantics, concurrent games


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