Switching, swapping, and replay

Issues for an open semantics for a *Java*-like calculus

E. Ábrahám M. Bonsangue F. S. de Boer A. Grüner M. Steffen

Christian-Albrechts University Kiel

AG Informatik, Logik, Mathematik

June 2005

Starting point

- question:

  what's observable of an open class-based, object-oriented, (multi-threaded) program

- $\rightsquigarrow$ compositional semantics
- component = "program fragment" = "open program"
- more details: later

Structure

Notion of observation

```
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 it run and see, whether the observer may be successful

- $P_1 \sqsubseteq_{may} P_2$: for all observers O: if $P_1 + O$ may be successful, then so may be $P_2 + O$.
- observational
  - “black-box”
  - fundamental distinction between program/component/player vs. environment/context/observer/opponent

Classes?

- open semantics (based on may testing): in principle: easy and understood

$\Rightarrow$ 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
  $\Rightarrow$ instantiation requests as interface interaction
  2. class = generators of object (via new)[1] $\Rightarrow$ replay
  3. abstraction of the heap topology

[1] Classes in *Java* or *C#* serve also as kind of types, and furthermore for inheritance. We ignore that mostly here.

What's hard for an open (f-a) semantics?

- "message passing"[2] framework $\Rightarrow$ in first approx.: semantics = message interchange at the interface
- open = environment absent/arbitrary

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

[2] no direct access to instance variables

What's hard for an open (f-a) semantics?

- "message passing"[2] framework $\Rightarrow$ in first approx.: semantics = message interchange at the interface
- open = environment absent/arbitrary

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

- well, depends . . .

[2] no direct access to instance variables

What's hard for an open (f-a) semantics?

- "message passing"[2] 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 $\in \mathit{Label}^*$?

[2] no direct access to instance variables

What's hard for an open (f-a) semantics?

- "message passing"[2] framework $\Rightarrow$ in first approx.: semantics = message interchange at the interface
- open = environment absent/arbitrary

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

- we know $P + O$ is a program of the language
  - well-formed
  - well-typed
  - class-structured
- exact representation
- $\Rightarrow$ formalization of those restrictions

[2] no direct access to instance variables

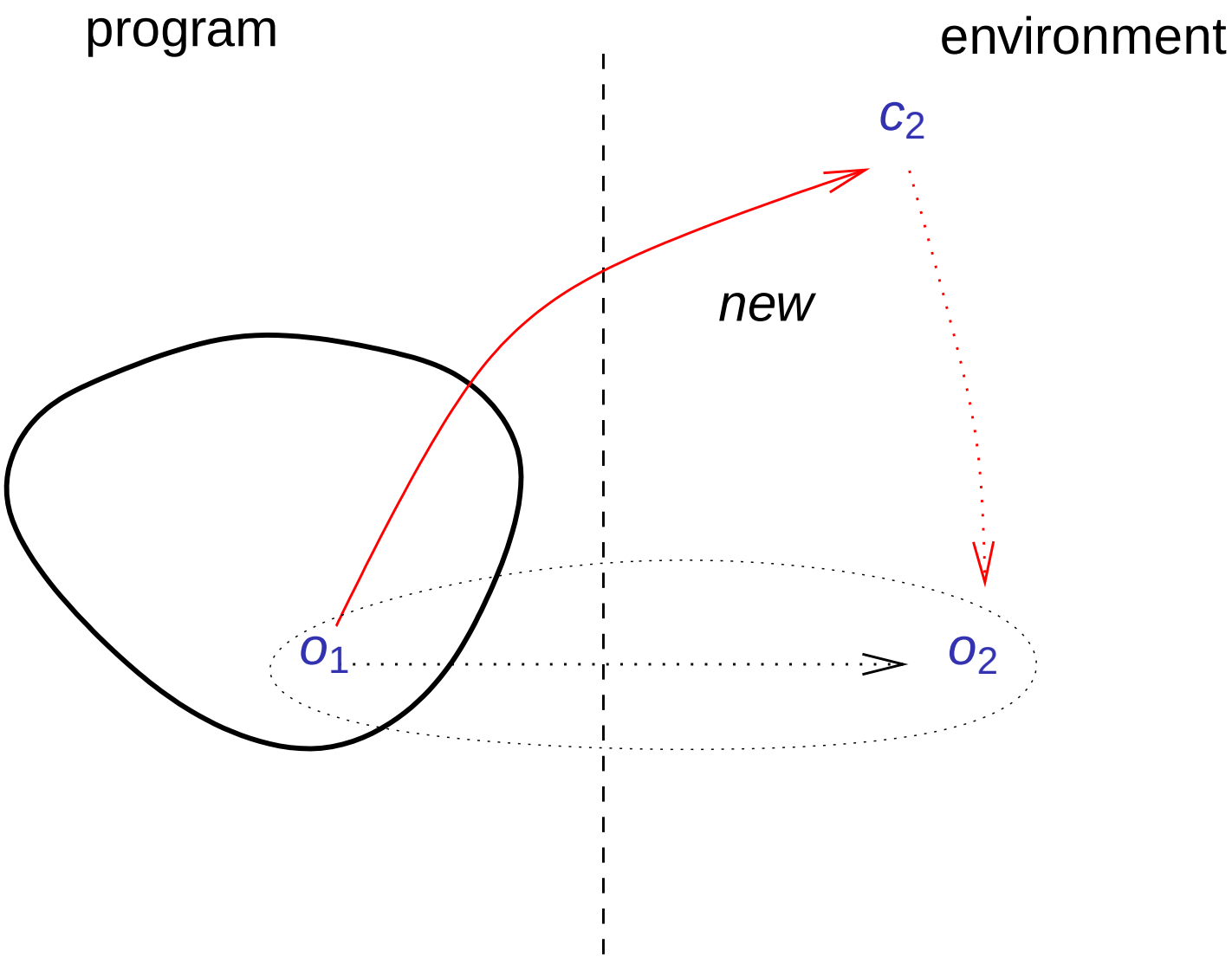
Open semantics

- operational description:
- assumption/commitment formulation
- $Ass. \vdash C : Comm. \xrightarrow{a} Ass'. \vdash C' : Comm'.$
- interface: 3 orthogonal abstractions:
  - static abstraction: type system
  - dynamic abstraction of heap topology:
  - abstraction of the stack structure of thread(s): enabledness conditions

Cross-border instantiation & heap abstraction

- classes as unit of code/exchange
- instantiation as interface interaction
- component instantiates observer class $\Rightarrow$
  - instance: part of the observer
  - reference to it: kept at the component

Cross-border instantiation & heap abstraction



Open semantics and heap abstraction

- exact interface behavior

$\Rightarrow$ abstraction of the heap topology necessary

- keep book about "whom it told what":

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

- assumption context: $E_\Delta \subseteq \Delta \times (\Delta + \Theta)$ = pairs of objects
- written $o_1 \hookrightarrow o_2$:
- worst case: equational theory implied by E_Δ (on Δ):

$$E_\Delta \vdash o_1 \leftrightharpoons o_2$$

(for $o_2 \in \Theta$: $E_\Delta \vdash o_1 \leftrightharpoons ; \hookrightarrow o_2$)

Dynamic heap abstraction

- partitioning of the heap: equivalence classes (“cliques”) of objects
- transition: change of contexts
- dynamicity
  - creation of new cliques
  - merge of existing cliques

Dynamic heap abstraction

- partitioning of the heap: equivalence classes (“cliques”) of objects
- transition: change of contexts
- dynamicity
  - creation of new cliques
  - merge of existing cliques
- outgoing communication
  - $a = n\langle call\ o_{receiver}.l(\vec{v})\rangle!$

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

  - update: $\acute{E}_\Delta = E_\Delta + \quad o_{receiver} \hookrightarrow \vec{v}$

Dynamic heap abstraction

- partitioning of the heap: equivalence classes ("cliques") of objects
- transition: change of contexts
- dynamicity
  - creation of new cliques
  - merge of existing cliques
- incoming communication
  - $a = n\langle call\ o_{receiver}.l(\vec{v})\rangle?$

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

  - check:[3] $E_\Delta \vdash o_{sender} \hookrightarrow \vec{v}$

[3] actually, it's $\acute{E}_\Delta$ instead of E_Δ.

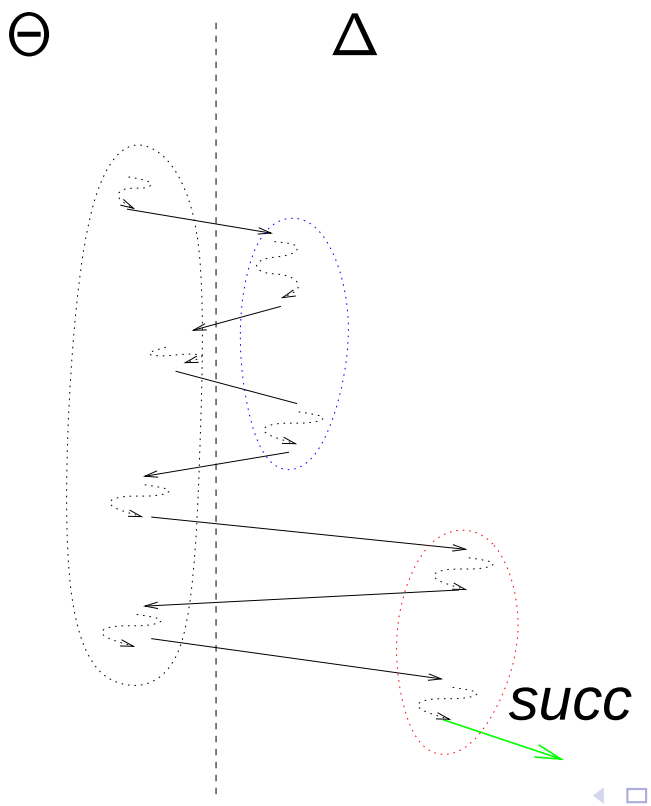
Where are we?

- open semantics in the presence of classes $\Rightarrow$ abstraction of heap topology
- features (*Java*/*C*$^{\#}$-inspired):
  - objects and classes (you might have guessed)
  - (multiple) threads
  - references/heap/aliasing
  - typed language
- formalized in some “object calculus”

Remember: observational /may-testing approach approach:

Two observers

- the observer is itself divided into cliques
- but: only one reports succcess
- consider P_1 on the left, interacting with two observers
- What does $P_1 \sqsubseteq_{may} P_2$ imply for P_2?



Two observers

- the observer is itself divided into cliques
- but: only one reports succcess
- consider P_1 on the left, interacting with two observers
- What does $P_1 \sqsubseteq_{may} P_2$ imply for P_2?

Two observers

- the observer is itself divided into cliques
- but: only one reports succcess
- consider P_1 on the left, interacting with two observers
- What does $P_1 \sqsubseteq_{may} P_2$ imply for P_2?

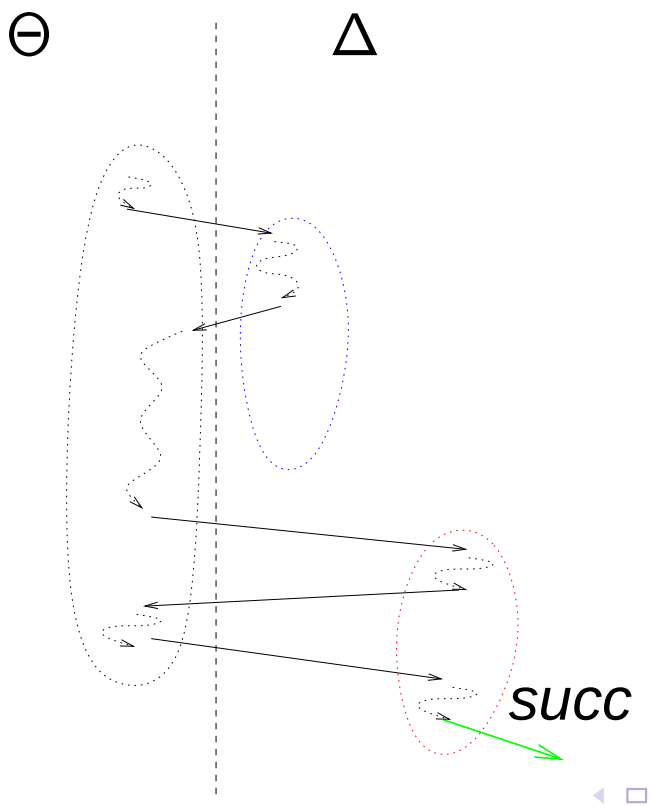
```
public class P1 {                                   // component
    public static void main(String[] arg) {
        O x1 = new O();
        x1.m1();
        O x2 = new O();
        x1.m2();
    }
}

class O {                                           // environment
    public void m1() { }
    public void m2() {
        System.out.println("success");
    }

}
```

Two observers

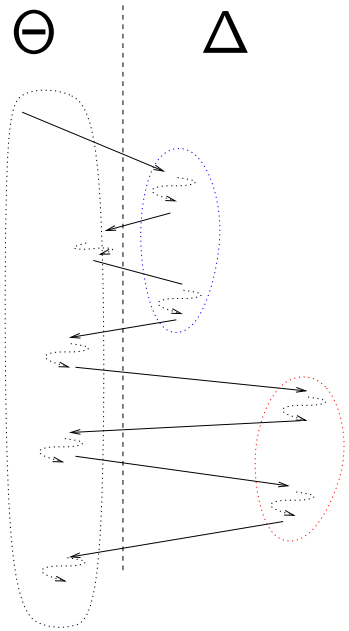
- the observer is itself divided into cliques
- but: only one reports succcess
- consider P_1 on the left, interacting with two observers
- What does $P_1 \sqsubseteq_{may} P_2$ imply for P_2?



Order of events

- separate observer cliques
- separate observer cliques cannot cooperate

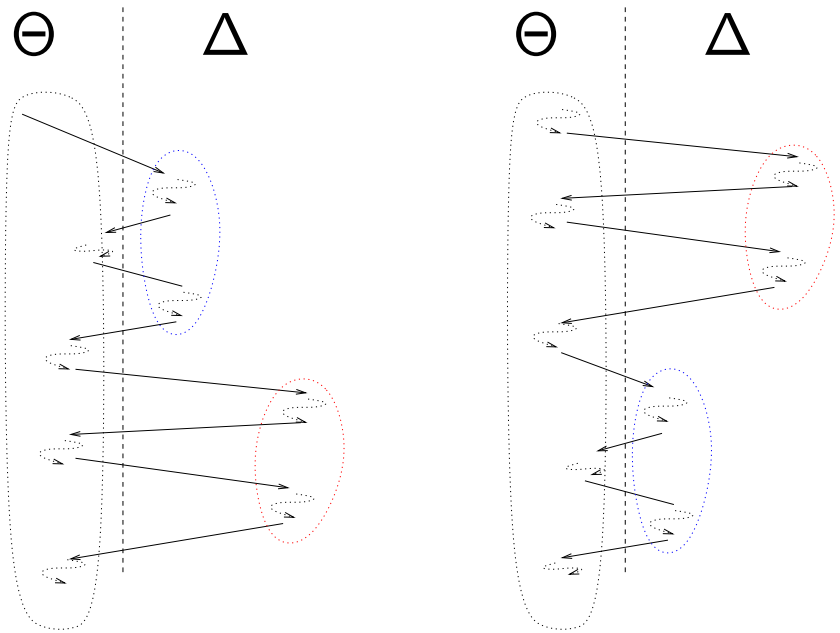
$\Rightarrow$ order of interaction not globally observable



Order of events

- separate observer cliques
- separate observer cliques cannot cooperate

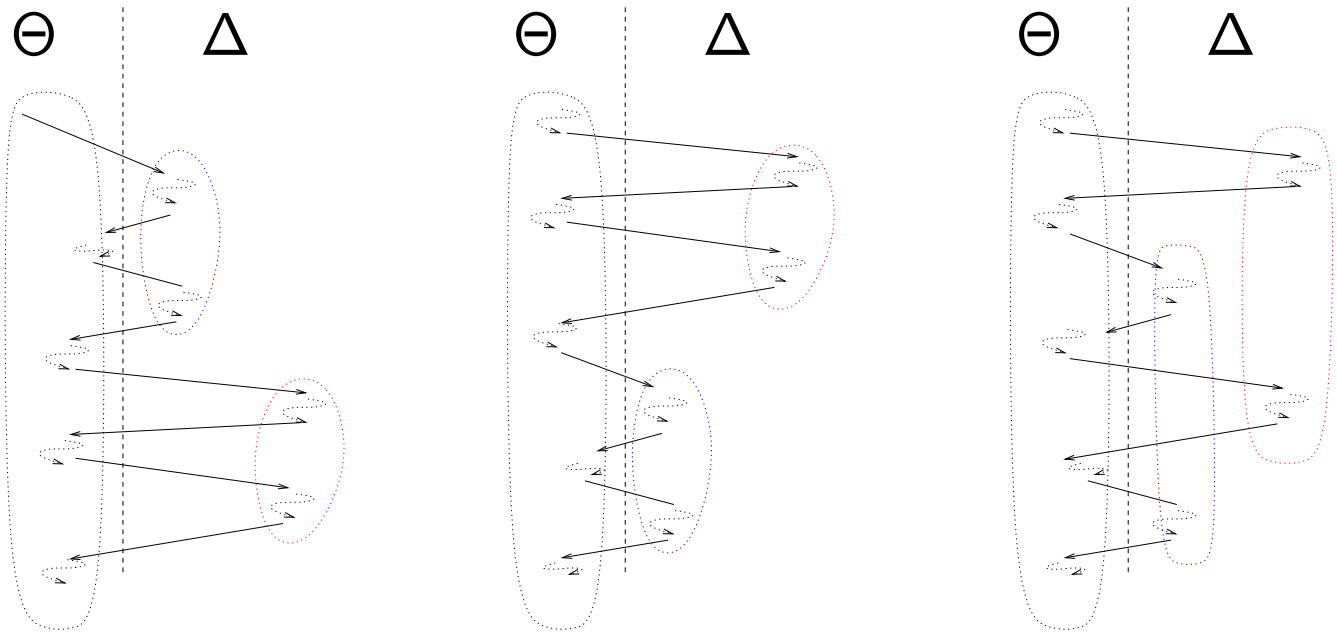
$\Rightarrow$ order of interaction not globally observable



Order of events

- separate observer cliques
- separate observer cliques cannot cooperate

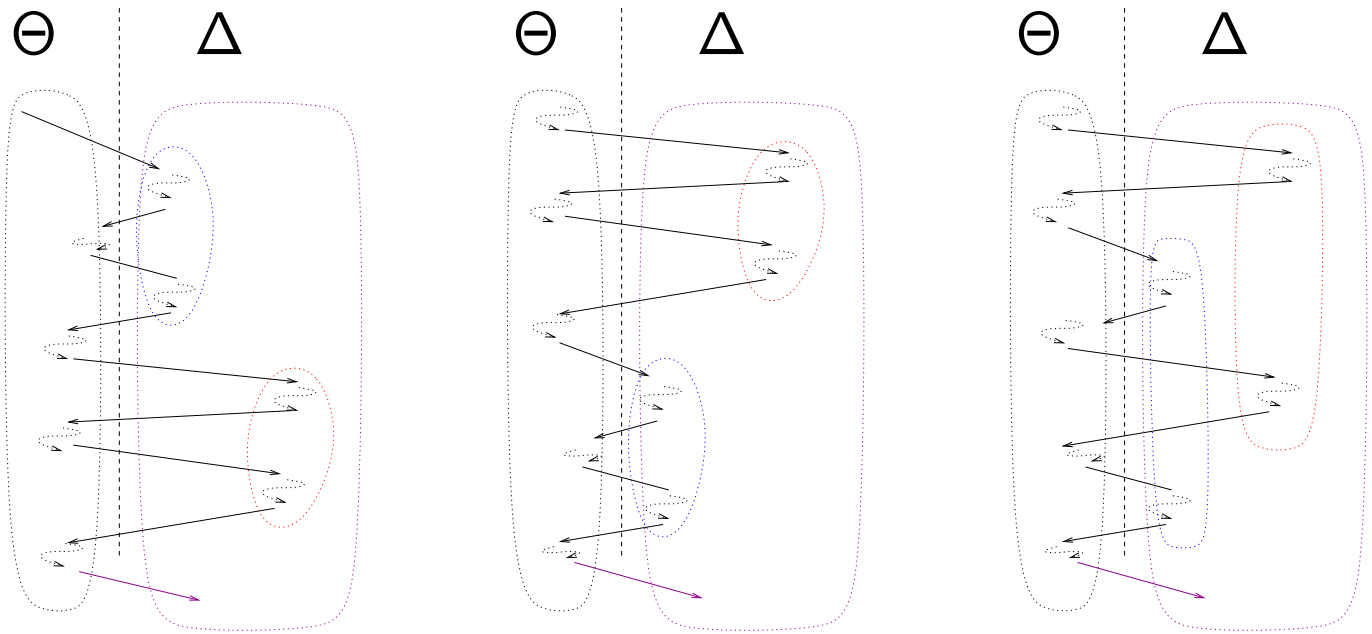
$\Rightarrow$ order of interaction not globally observable



Order of events

- separate observer cliques
- separate observer cliques cannot cooperate

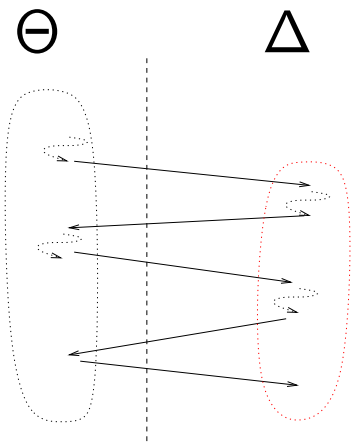
$\Rightarrow$ order of interaction not globally observable



Classes as generators of objects

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

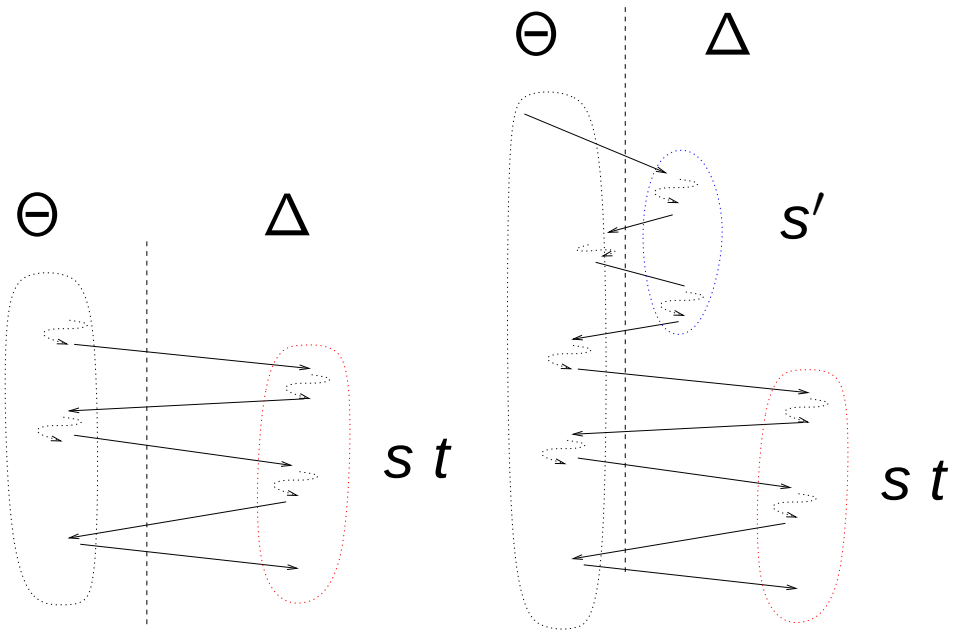
  what can be observed once by one observer clique, can be observed again (up-to identity) by a second "instance" of the observer



Classes as generators of objects

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

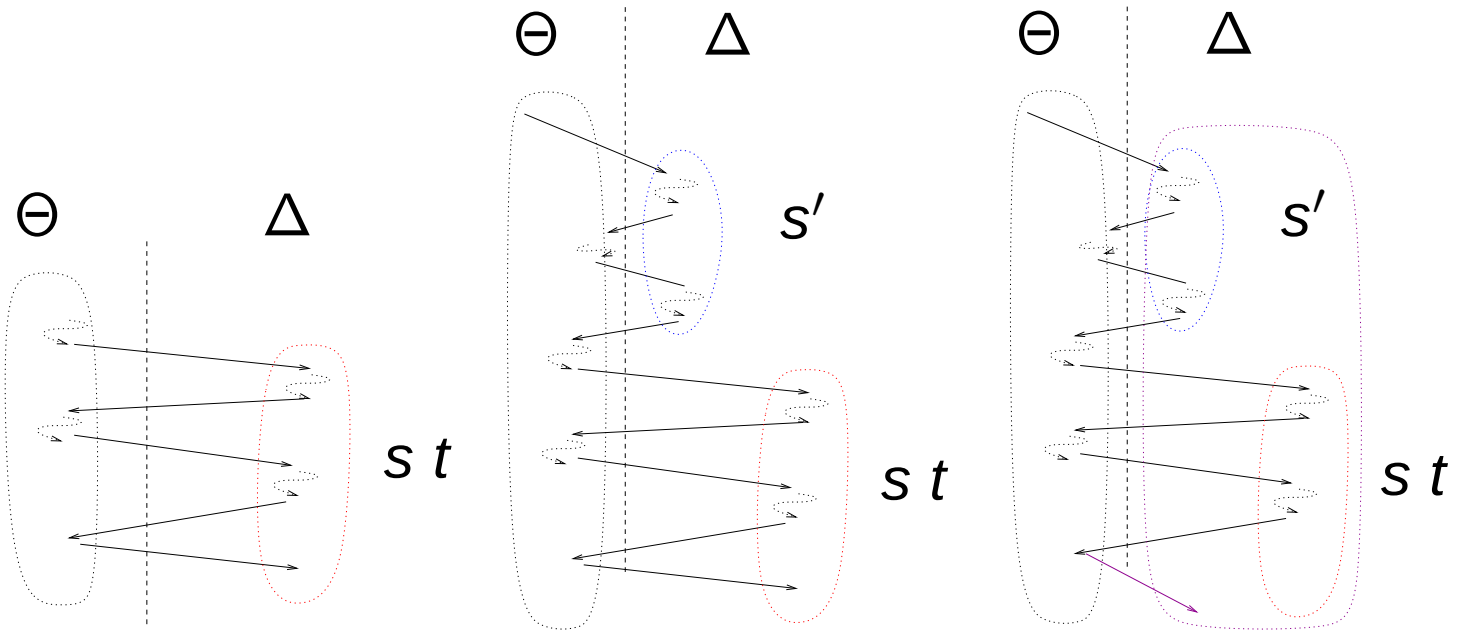
  what can be observed once by one observer clique, can be observed again (up-to identity) by a second "instance" of the observer



Classes as generators of objects

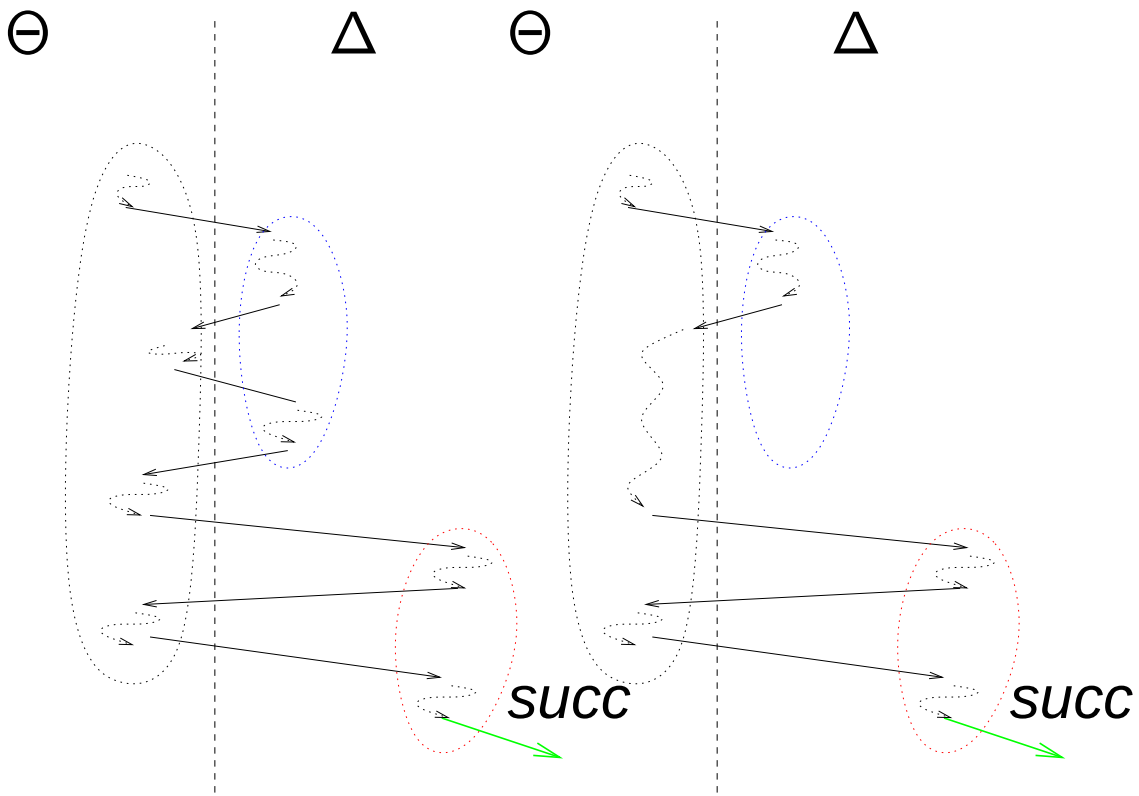
- two new instances of a class are identical up-to their id
- for the observer:

  what can be observed once by one observer clique, can be observed again (up-to identity) by a second "instance" of the observer



Two observers, revisited

- observer cliques are independent
- consider again the first examples: 2 cliques

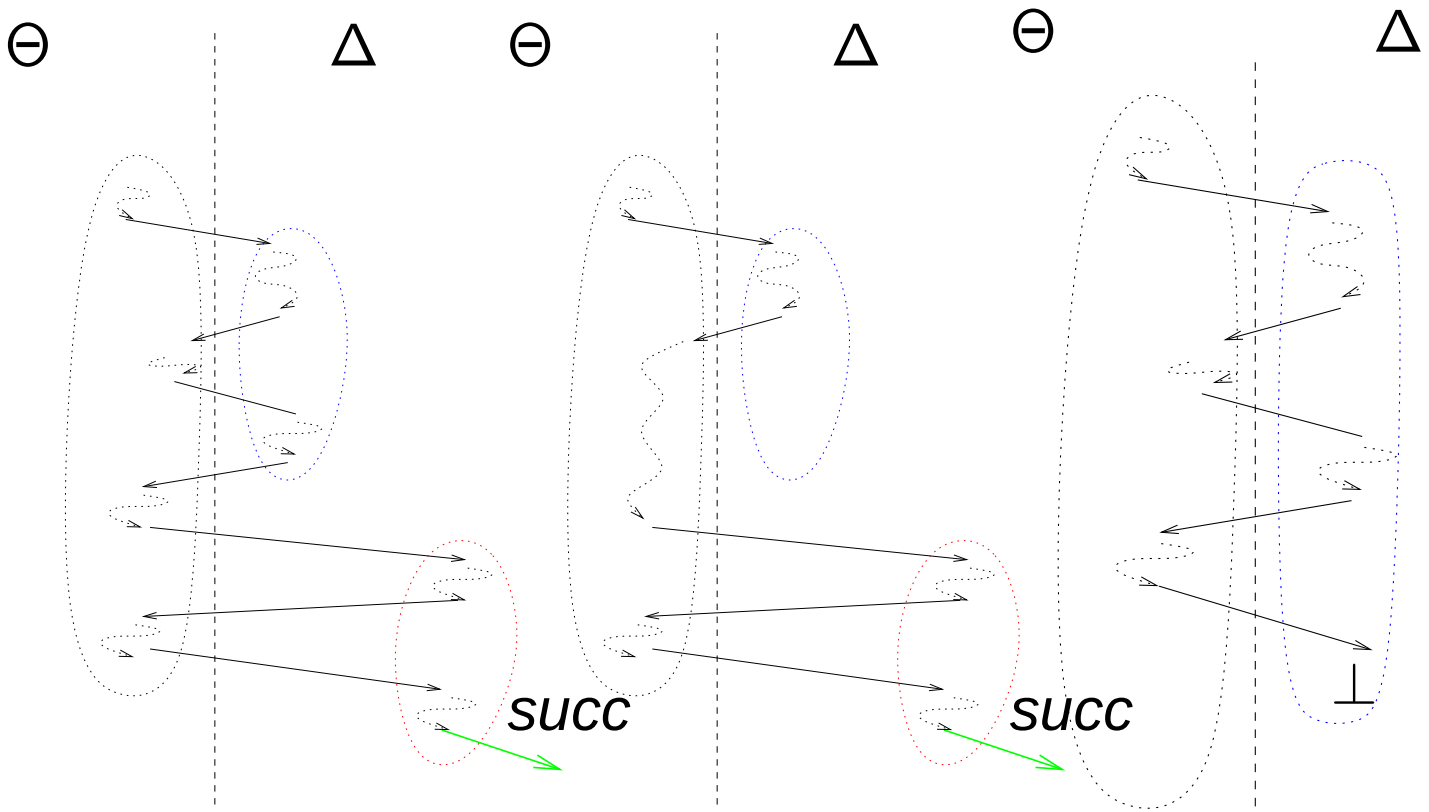


Two observers, revisited

- observer cliques are independent
- consider again the first examples: 2 cliques
- of course, another observer may test for the “first interaction”
- does it mean: only “trace” per clique? (projection)
- reason(?): no information can be passed from the first to the 2nd observer clique

Two observers, revisited

- observer cliques are independent
- consider again the first examples: 2 cliques



Two observers, revisited

- observer cliques are independent
- consider again the first examples: 2 cliques
- an observer reporting success, could additionally observe, that the interaction with the other clique is a prefix of the original, but not longer

Results

- full-abstraction for may-testing in some object-calculus setting with classes
- calculus
  - strongly typed, nominal types
  - multi-threaded
  - name-generation
  - algebraic formulation (“object calculus”)
- semantics (formalizing the ideas sketched here):
  - scope extrusion mechanism to deal with object identities
  - acquaintance as (dynamic) equivalence relation between objects
  - equivalence relation on traces to capture independence of order
  - characterization of swapping, switching, and replay

Results

Definition ($\sqsubseteq_{trace}$)

$\Xi_0 \vdash C_1 \sqsubseteq_{trace} C_2$, if the following holds. For all $\Xi_0 \vdash C_1 \stackrel{t}{\Longrightarrow}$ and all environment cliques $[o_t]$ after t, there exists $\Xi_0 \vdash C_2 \stackrel{s}{\Longrightarrow}$ such that

1. there exists an environment clique $[o_s]$ after s such that $\Xi_0 \vdash s \downarrow_{[o_s]} \asymp_\Delta t \downarrow_{[o_a]}$, and
2. $\Xi_0 \vdash t \lessapprox_\Delta s$.

- $\asymp_\Delta$: up-to swapping (and switching)
- $\lessapprox_\Delta$: up-to swapping, replay, prefix (and switching)

Multi-threading

- Note: (most) everything I told so far was not depending on concurrency
- introduction of concurrency (=“multithreading”)
  - conceptually not complex
  - threads themselves “do not communicate”: all information transfer “via objects”
  - introduction of names for threads + thread name into the communication labels
  - definability/completeness proof requires “implementation” of (distributed) “mutex”-algorithm

Determinism

- single-threaded setting
- not (!) uniformey a simplification
- classes as generators of objects (“replay”)
  - a single (!) trace may be deterministic or *non-deterministic*
  - characterization of deterministic traces required
- deterministic:

$$\textit{same history} \rightsquigarrow \textit{same response}$$

- note:
  - history per clique
  - history up-to equivalences (swapping, switching etc)

Class variables

- discussion so far: instance variables, only

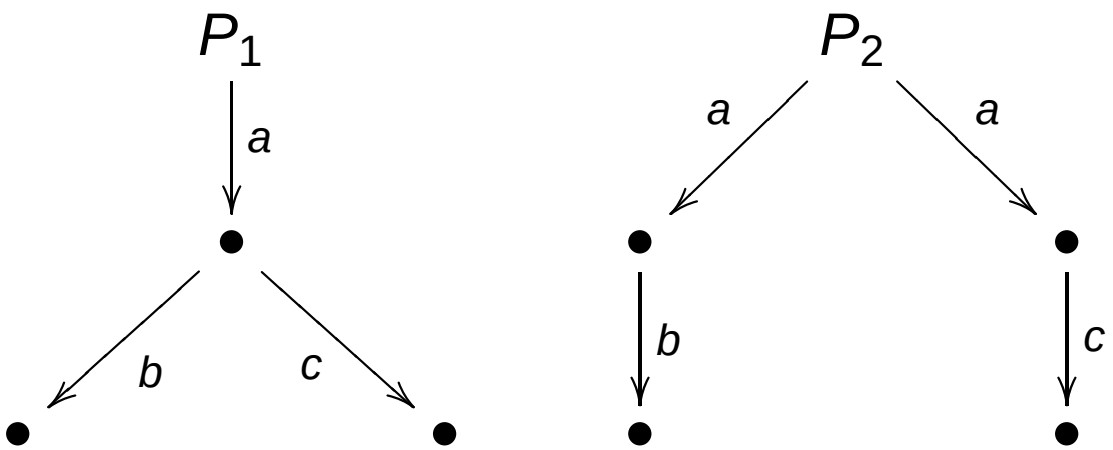
$\Rightarrow$ different instances of the same class are identical up-to identity: replay

- class-variables: 2 important consequences
  - allows to distinguish different instances $\Rightarrow$ replay-phenomenon no longer relevant
  - provide a communication channel between various instances of the class $\Rightarrow$ all instance of a class are connected

Cloning

- creates a “identical copy” up-to identity
- `new` = “clone of the initial state”
- makes the branching structure visible

Cloning



Cloning

```
public class O {   // component
  public static void main(String[] arg) {
    P1 x = new P1();
    P1 y;
    x.a();
    y = (P1)x.clone();
    x.b();y.c();
    System.out.println("success");
  }
}
```

Cloning

```
class P1 implements Cloneable {
  private int x = 0;
  private java.util.Random gen = new java.util.Random();

  public Object clone ()  {
    try {return super.clone();}            // use the native clone-method
    catch(CloneNotSupportedException e) {  // just catch it.
    }
    return new P2();                       // unreachable
  }

  public void choose () { x=gen.nextInt(2)+1;return;}    // x in {1,2}

  public void  a() {return;}
  public void  b() {
    this.choose();
    if (x==1) {return;} else {System.exit(0);};
  }
  public void  c() {
    this.choose();
    if (x==2) {return;} else {System.exit(0);}
  }
}
```

Thread classes

- classes = generator of state
- “thread class” = generator of activity
- cross-border thread spawning

Subclassing

- “opens up” a new interface

$\Rightarrow$ new observations possible by subclassing

- most important: overriding makes “self-communication” observable

Subclassing

```
class P1  {
    void add () {...}          // adds one
    void add2 () {... }        // adds two
    }

class P2  {
    void add () {...}                          // adds one
    void add2 () {... self.add() ... }         // adds two
    }

class O extends P<x> {
    add() { .... }                          // overriding
    ...

}
```

Conclusions

- are classes good composition units?
- on the agenda:
  - (fully) compositional semantics (under work)
  - trace logics
  - delegation, subtyping (and subclassing), cloning, generics
    ...
- game semantics

References I

[1] E. Ábrahám, M. M. Bonsangue, F. S. de Boer, and M. Steffen.
Object connectivity and full abstraction for a concurrent calculus of classes.
In Z. Li, editor, *ICTAC'04*, volume 3407 of *Lecture Notes in Computer Science*, pages 38–52. Springer-Verlag, July 2004.

[2] E. Ábrahám, M. M. Bonsangue, F. S. de Boer, and M. Steffen.
Object connectivity for a concurrent class calculus (extended abstract).
2004.
Submitted for publication. A preliminary and longer version appeared under the title "A Structural Operational Semantics for a Concurrent Class Calculus" as Technical Report 0307, CAU, Institute of Computer Science August 2003.

[3] E. Ábrahám, F. S. de Boer, M. M. Bonsangue, A. Grüner, and M. Steffen.
Observability, connectivity, and replay in a sequential calculus of classes.
In M. Bosangue, F. S. de Boer, W.-P. de Roever, and S. Graf, editors, *Proceedings of the Third International Symposium on Formal Methods for Components and Objects (FMCO 2004)*, Lecture Notes in Computer Science. submitted for publication, 2005.
To appear.

[4] E. Ábrahám, A. Grüner, and M. Steffen.
Dynamic heap-abstraction for open, object-oriented systems with thread classes.
May 2005.
Submitted as conference contribution.

[5] E. Ábrahám, A. Grüner, and M. Steffen.
An open structural operational semantics for an object-oriented calculus with thread classes.
Technical Report 0505, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, May 2005.