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
introduction

classes and observable behavior

4 complications

results

variations

conclusion
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

introduction

classes and observable behavior

4 complications

results

variations

conclusion
Notion of observation

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

classes and observable behavior

4 complications

results

variations

conclusion
Classes?

- open semantics (based on 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)\(^1\) ⇒ 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 ⇒ in first approx.: semantics = message interchange at the interface
- open = environment absent/arbitrary
⇒ 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 \(\in Label^*\) ?

\(^2\)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

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

- **interface:** 3 orthogonal abstractions:
  - static abstraction: *type* system
  - dynamic abstraction of *heap topology*:
  - abstraction of the *stack* structure of thread(s): *enabledness* conditions
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

Introduction

Classes and observable behavior

4 complications results variations conclusion

Cross-border instantiation & heap abstraction

program

environment

new

O₁

O₂
Open semantics and heap abstraction

- exact interface behavior
  ⇒ 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 \( o_1 \leftrightarrow o_2 : \)
- worst case: equational theory implied by \( E_\Delta \) (on \( \Delta \)):

\[ E_\Delta \vdash o_1 \leftrightarrow o_2 \]

(for \( o_2 \in \Theta: E_\Delta \vdash o_1 \leftrightarrow; \leftrightarrow o_2 \))
Dynamic heap abstraction

- **partitioning** of the heap: equivalence classes ("cliques") of objects
- **transition**: change of contexts
- **dynamincity**
  - creation of new cliques
  - merge of existing cliques
Dynamic heap abstraction

- **partitioning** of the heap: equivalence classes ("cliques") of objects
- transition: change of contexts
- **dynamicy**
  - creation of new cliques
  - merge of existing cliques
- **outgoing** communication
  - \[ a = n\langle \text{call } o_{\text{receiver}}.l(\vec{v})\rangle! \]

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

- **update**: \( \hat{E}_\Delta = E_\Delta + o_{\text{receiver}} \leftrightarrow \vec{v} \)
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
  - \( a = n\langle \text{call } o_{\text{receiver}} . l(\vec{v}) \rangle \)?

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

- check:\(^3 \) \( E_\Delta \vdash o_{\text{sender}} \leftrightarrow \vec{v} \)

\(^3\) actually, it’s \( \dot{E}_\Delta \) instead of \( E_\Delta \).
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*
introduction

classes and observable behavior

4 complications

results

variations

conclusion
Two observers

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

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

```java
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 success
- 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

⇒ order of interaction not globally observable
Order of events

- separate observer cliques
- separate observer cliques cannot cooperate

⇒ order of interaction not globally observable
Order of events

- separate observer cliques
- separate observer cliques cannot cooperate

⇒ order of interaction not globally observable
Order of events

- separate observer cliques
- separate observer cliques cannot cooperate
  \[\Rightarrow \text{order of interaction not globally observable}\]
Classes as generators of objects

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

\[ \text{what can be observed once by one observer clique, can be observed again (up-to identity) by a second “instance” of the observer} \]
Two observers, revisited

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

\[ \Theta \quad \Delta \quad \Theta \quad \Delta \]

\[ \Theta \quad \Delta \quad \Theta \quad \Delta \]

\[ succ \quad succ \]

AG, June 2005
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
introduction

classes and observable behavior

4 complications

results

variations

conclusion
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** ($\sqsubseteq_{\text{trace}}$)

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

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

2. $\Xi_0 \vdash t \preceq_{\Delta} s$.

- $\sim_{\Delta}$: up-to swapping (and switching)
- $\preceq_{\Delta}$: up-to swapping, replay, prefix (and switching)
introduction

classes and observable behavior

4 complications

results

variations

conclusion
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 (!) uniformly a simplification
- classes as generators of objects ("replay")
  - a single (!) trace may be deterministic or non-deterministic
  - characterization of deterministic traces required
- deterministic:
  
  \[ \text{same history} \sim \text{same response} \]

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

- discussion so far: instance variables, only
  - different instances of the same class are identical up-to identity: replay
- class-variables: 2 important consequences
  - allows to distinguish different instances ⇒ replay-phenomenon no longer relevant
  - provide a communication channel between various instances of the class ⇒ 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

$P_1$

$P_2$

Switching, swapping, and replay
Cloning

```java
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
  - new observations possible by subclassing
- most important: overriding makes “self-communication” observable
Subclassing

```java
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
    ...
}
```

AG, June 2005  Switching, swapping, and replay
introduction
classes and observable behavior
4 complications
results
variations

conclusion
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

Object connectivity and full abstraction for a concurrent calculus of classes.

Object connectivity for a concurrent class calculus (extended abstract).
2004.

Observability, connectivity, and replay in a sequential calculus of classes.
To appear.

Dynamic heap-abstraction for open, object-oriented systems with thread classes.
May 2005.
Submitted as conference contribution.

An open structural operational semantics for an object-oriented calculus with thread classes.