

Inheritance & Observability

Erika Ábrahám and Martin Steffen

RWTH Aachen University of Oslo

FM Seminar Oslo

May 3, 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

sequential programs	⇒	state-transformers, continuous functions
concurrency	⇒	
objects	⇒	
classes	⇒	
locks/monitors	⇒	
cloning	⇒	

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

sequential programs	⇒	state-transformers,	continu-
		ous functions	
concurrency	⇒	“traces”	
objects	⇒	“traces”	
classes	⇒	“connectivity”,	abstract
		heap	
locks/monitors	⇒		
cloning	⇒		

Language features

sequential programs	⇒	state-transformers, continuous functions
concurrency	⇒	“traces”
objects	⇒	“traces”
classes	⇒	“connectivity”, abstract heap
locks/monitors	⇒	tricky dependencies to capture mutex
cloning	⇒	

Language features

sequential programs	⇒	state-transformers, continuous functions
concurrency	⇒	“traces”
objects	⇒	“traces”
classes	⇒	“connectivity”, abstract heap
locks/monitors	⇒	tricky dependencies to capture mutex
cloning	⇒	[“branching”]

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();  
    }  
}
```

Challenges & variation points

Bottom line

“inheritance breaks encapsulation”

- “exact” characterization of the interface behavior
- ingredients here:
 - dynamic creations of “entities”, especially objects
 - lazy instantiation
 - *connectivity*
 - irrelevance of object *identities*
 - *replay*
 - concurrency¹ and *asynchronicity*
- life made easier by: *no re-entrance*

¹Sometimes, concurrency makes life easier. . . . 

Variation points

- “private” vs. public fields
- private vs. public methods
- super-keyword
- “shadowing”: binding of methods vs. binding of

Fields and shadowing

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

```
class C2 extends C1 {  
    x; // overriding/shadowing  
    n () { ... m() ... }  
}
```

Accessor methods

```
class C1 {  
  x;  
  getX() { x }  
  m () { .. self.getX()... }  
}
```

```
class C2 extends C1 {  
  x;  
  getX() { x }  
  n () { ... m() ... }  
}
```

Private vs. public methods

```
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

Can one observe the **run-time type**

let $x:C_1 = \text{new } C_1$ vs. let $x:C_1 = \text{new } C_2$ (1)

Observability of self-calls

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

- closed and *open* semantics
- embedding vs. delegation for representing objects
- lazy instantiation

Language

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

Syntax

$C ::= \mathbf{0} \mid C \parallel C \mid \underline{v(n:T).C} \mid n[(O)] \mid \underline{n[M, F, L]} \mid \underline{n\langle t \rangle}$

$O ::= n, F, L$

$M ::= l = m, \dots, l = m$

$F ::= l = f, \dots, l = f$

$m ::= \zeta(n:T).\lambda(x:T, \dots, x:T).t$

$f ::= \zeta(n:T).\lambda().v \mid \zeta(n:T).\lambda().\perp_{n'}$

$t ::= v \mid \text{stop} \mid \text{let } x:T = e \text{ in } t$

$e ::= t \mid \text{if } v = v \text{ then } e \text{ else } e \mid \text{if } \text{undef}(v.l()) \text{ then } e \text{ else } e$
| $o@l(\vec{v}) \mid v.l() \mid v.l := \zeta(s:n).\lambda().v$
| $\text{new } n \mid \text{claim}@n \mid \underline{\text{get}@n} \mid \text{suspend}(n) \mid \underline{\text{grab}(n)} \mid \underline{\text{release}(n)}$

$v ::= x \mid n \mid ()$

$L ::= \perp \mid T$

com
obje
met
field
met
field
thre
exp

valu
lock

Open semantics

- environment vs. component
- **assumption** / *commitment* formulation
 - commitment = component sides
 - assumptions = abstract representation of

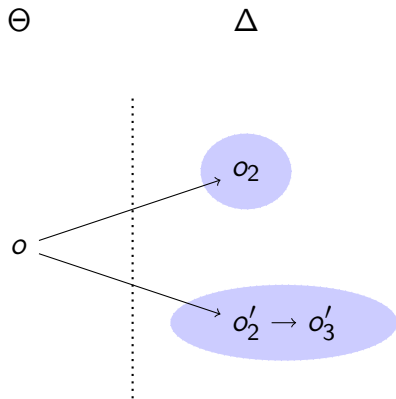
$$\Gamma \vdash C : \Theta \xrightarrow{\text{"label"}} \Gamma' \vdash C' : \Theta'$$

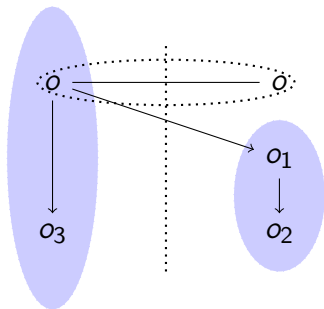
- assumption/commitments
 - 1 typing
 - 2 available objects, threads
 - 3 **lock** status
 - 4 **connectivity**

Connectivity

Worst-case approximation of “who may know who” in the environment (resp. component) part of the heap.

Cross-border inheritance



Θ Δ 

Dynamic binding and embedding

- 1 *private* members all named differently
- 2 *embedding*: methods copied in

$$\frac{\Gamma \vdash \text{find}(c) = F, M}{\Gamma \vdash n\langle \text{let } x: T = \text{new } c \text{ in } t \rangle \rightsquigarrow \Gamma \vdash v(o:c).(o[F, M, \perp] \parallel n\langle \text{let } x: T = o \text{ in } t \rangle)} \text{NewO}$$

$$\frac{\Gamma \vdash c = \llbracket (\perp, F, M) \rrbracket}{\Gamma \vdash \text{find}(c) = F, M} \text{F-Top}$$

$$\frac{\Gamma \vdash c_1 = \llbracket (c_2, F_1, M_1) \rrbracket \quad \Gamma \vdash \text{find}(c_2) = F_2, M_2 \quad M = M_1, M_2 \setminus M_1 \quad F = F_1, F_2}{\Gamma \vdash \text{find}(c_1) = F, M} \text{F-I}$$

Legal traces

- existential abstraction of the component, as well
- traces, which are possible “at all”
- $\Gamma \vdash t : \text{trace} :: \Theta$

References I

- [Abadi and Cardelli, 1996] Abadi, M. and Cardelli, L. (1996).
A Theory of Objects.
Monographs in Computer Science. Springer-Verlag.
- [Ábrahám et al., 2004] Ábrahám, E., Bonsangue, M. M., de Boer, F. S., and Steffen, M. (2004).
Object connectivity and full abstraction for a concurrent calculus of classes.
In Li, Z. and Araki, K., editors, *ICTAC'04*, volume 3407 of *Lecture Notes in Computer Science*, pages 37–51. Springer-Verlag.
- [Ábrahám et al., 2005] Ábrahám, E., de Boer, F. S., Bonsangue, M. M., Grüner, A., and Steffen, M. (2005).
Observability, connectivity, and replay in a sequential calculus of classes.
In Bonsangue, M., de Boer, F. S., de Roeper, W.-P., and Graf, S., editors, *Proceedings of the Third International Symposium on Formal Methods for Components and Objects (FMCO 2004)*, volume 3657 of *Lecture Notes in Computer Science*, pages 296–316. Springer-Verlag.
- [Ábrahám et al., 2009] Ábrahám, E., Grabe, I., Grüner, A., and Steffen, M. (2009).
Behavioral interface description of an object-oriented language with futures and promises.
Journal of Logic and Algebraic Programming, 78(7).
Special issue of the *Journal of Logic and Algebraic Programming* with selected contributions of NWPT'07. The paper is a reworked version of an earlier UiO Technical Report TR-364, Oct. 2007.
- [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.
- [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.

References II

[Steffen, 2006] Steffen, M. (2006).

Object-Connectivity and Observability for Class-Based, Object-Oriented Languages.
Habilitation thesis, Technische Fakultät der Christian-Albrechts-Universität zu Kiel.

[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.