Towards full abstraction for class-based, multithreaded OO

— Work in progress —

München, February 2003

Overview

- introduction, full-abstraction
- object-based and class-based calculus
- issues for full abstraction
- conclusion

Object & class based languages

- class-based oo
  - mainstream of oo (C^{++}, *Smalltalk*, *Java*, ...)
  - class as unit of code/reuse (inheritance) and (often) as unit of abstraction (type)
- object-based
  - no classes, no (class)-inheritance
  - dynamic method update

Full abstraction: starting point

- basically: comparison between 2 semantics, resp. 2 implied notions of equality
- given a reference semantics, the 2nd one is
  - neither too abstract = sound
  - nor too concrete = complete
- Milner [Mil77], Plotkin [Plo77] for λ-calculus/LCF
- various *variations* of the theme

Full abstraction: standard setup

- *reference* semantics:
  - must be natural
  - easy to define
  - non-compositional

$\Rightarrow$ $\boxed{\text{contextual, observational}}$

- context $\mathcal{C}[\_]$= "program with a hole"
- filling the hole with a part of a program (component C): complete program $\mathcal{C}[C]$
- what is a context/component?: depends on the language/syntax (sequential/parallel/functional ... contexts)

F-A: standard setup (cont'd)

- given a closed program P: $\mathcal{O}(P)$ = observations

$\Rightarrow$ observational equivalence:

$$c_1 \equiv_{obs} c_2 \quad \text{iff} \quad \forall \mathcal{C}.\ \mathcal{O}(\mathcal{C}[c_1]) = \mathcal{O}(\mathcal{C}[c_2])$$

- given a denotational semantics $[\![\_]\!]_{\mathcal{D}}$, resp. the implied equality $\equiv_{\mathcal{D}}$

$\Rightarrow$ $\equiv_{\mathcal{D}}$ is fully abstract wrt. $\equiv_{obs}$:

$$\equiv_{obs} \quad = \quad \equiv_{\mathcal{D}}$$

Object calculus: informal

- formal model(s) of oo languages
- in the tradition of the λ-calculi, process calculi ...
- more specifically:
  - object-calculi of Abadi/Cardelli [AC96]
  - π-calculus: processes, parallelism, name-passing [MPW92][SW01]
  - ν-calculus: λ-calc. with name creation (references) respectively its concurrent version [PS93][GH98]

Concurrent ν-calculus: Syntax

- program = “set” of named threads and objects running in parallel: $n\langle t\rangle$ and $n[l_1 = m_1, \ldots, l_k = m_k]$
- dynamic scoping of names
  - $\nu n{:}T.\ (C_1 \parallel C_2)$
  - ν acts as binder: α-equivalence
  - communication of names changes the scope (“scope extrusion”)
- methods = functions with specific “self”-parameter [a]
- active entities: threads
  - sequencing + local, static scoping: $let\ x = e\ in\ t$
  - thread creation

[a] In the presence of subtyping, the parameter would be late-bound.

Concurrent ν-calculus: Syntax

$$
\begin{array}{rcll}
P & ::= & \mathbf{0} \mid P \parallel P \mid \nu(n{:}T).P \mid n[O] \mid n\langle t\rangle & \text{components} \\
O & ::= & l = m, \ldots, l = m & \text{object} \\
m & ::= & \varsigma(n{:}T).\lambda(x{:}T, \ldots, x{:}T).t & \text{method} \\
t & ::= & v \mid stop \mid let\, x{:}T = e\; in\; t & \text{thread} \\
e & ::= & t \mid \mathsf{if}\; v = v\; \mathsf{then}\; e\; \mathsf{else}\; e & \text{expressions} \\
 & \mid & v.l(v, \ldots, v) \mid n.l \Leftarrow m & \\
 & \mid & new[O] \mid new\langle t\rangle \mid currentthread & \\
v & ::= & x \mid n & \text{values}
\end{array}
$$

Adding classes

- class: just like objects:
  - named collection of methods $n[\![l_1 = m_1, \ldots]\!]$
  - instantiated by name, not structure: $new\, n$!
  - class names are not first-order citizens, i.e., not subject to
    - ν-binding (= hiding)
    - storing, sending, receiving etc.
- method update not used in class-based setting

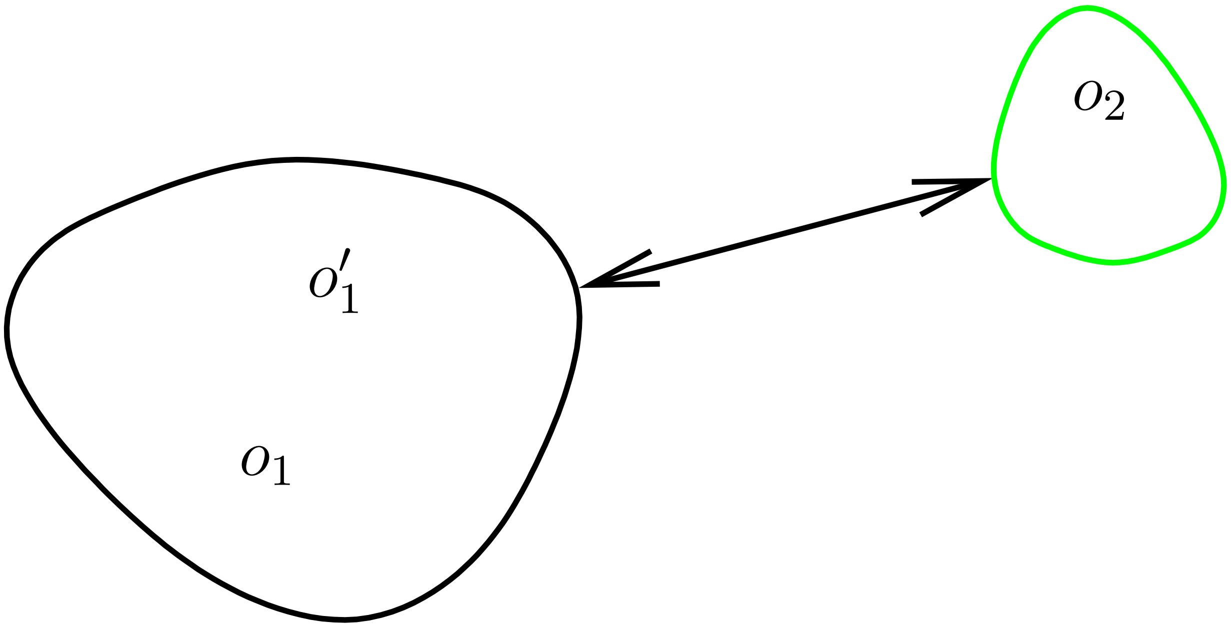
Adding classes

$$
\begin{array}{rcll}
P & ::= & \mathbf{0} \mid n[\![O]\!] \mid n\langle\!\langle t \rangle\!\rangle \mid P \parallel P \mid R & \text{program (stat.)} \\
R & ::= & \mathbf{0} \mid R \parallel R \mid \nu(n{:}T).R \mid n[O] \mid n\langle t \rangle & \text{program (dyn.)} \\
O & ::= & l = m, \ldots, l = m & \text{object} \\
m & ::= & \varsigma(n{:}T).\lambda(x{:}T, \ldots, x{:}T).t & \text{method} \\
t & ::= & v \mid stop \mid let\, x{:}T = e\; in\; t & \text{thread} \\
e & ::= & t \mid \textsf{if}\; v = v\; \textsf{then}\; e\; \textsf{else}\; e \mid v.l(v, \ldots, v) & \text{expression} \\
 & \mid & new\, n \mid new\, n\langle t \rangle & \\
v & ::= & x \mid n & \text{values}
\end{array}
$$

Semantics (1)

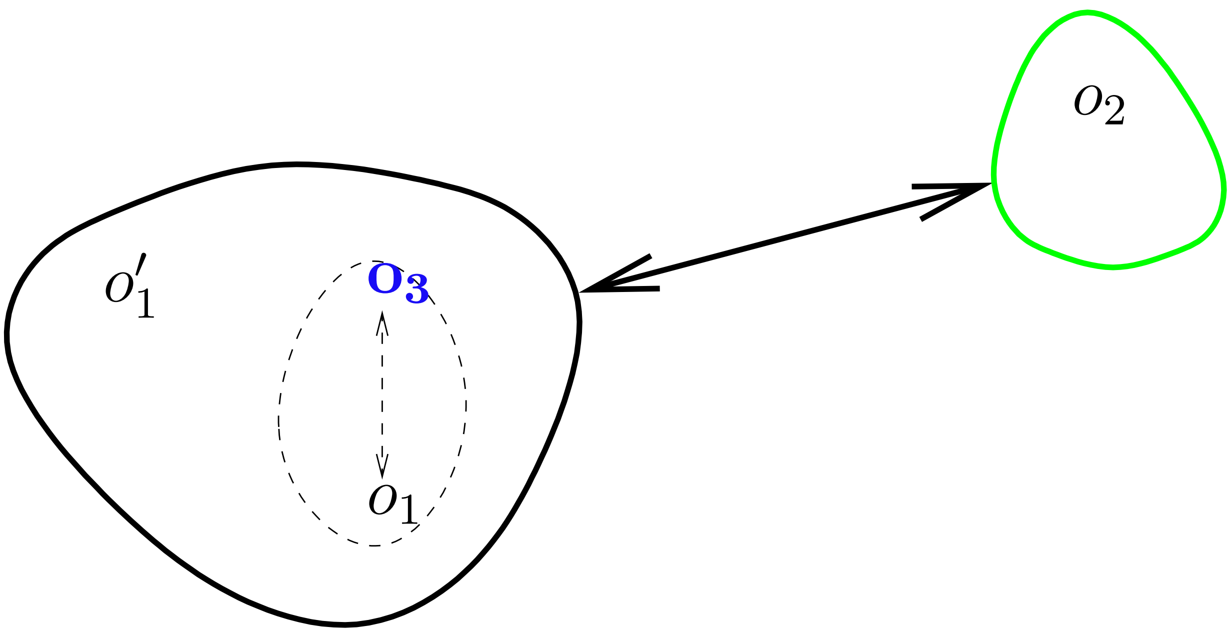
- given in various “stages”
  - internal (configuration-local) steps
  - external, global steps, interacting with the environment
  - computation steps modulo α-conversion
- typed operational semantics

Internal steps



- black: objects of the component
- green: objects of the environment

Internal steps



- o_1 creates an internal object $\mathbf{o_3}$ (assume: thread n visits o_1)

$$\mathbf{c}[\![(O)]\!] \parallel n\langle let\, x{:}T = new\, \mathbf{c}\; in\; t\rangle \rightsquigarrow$$
$$c[\![(O)]\!] \parallel \nu\mathbf{o_3}{:}\mathbf{T}.\, (n\langle let\, x{:}T = \mathbf{o_3}\; in\; t\rangle \parallel \mathbf{o_3}[\mathbf{O}].\,)$$

Semantics: Internal steps

- 4 exemplary axioms
- confluent ($\rightsquigarrow$) and non-confluent ($\xrightarrow{\tau}$) internal steps
- for CALL$_i$: $O.l(o)(\vec{v})\ in\ t$: parameter passing, and especially replacing the ς-bound self-parameter by o.

$$n\langle let\ x{:}T = \mathbf{v}\ in\ t\rangle \rightsquigarrow n\langle t[\mathbf{v}/\mathbf{x}]\rangle \qquad \text{RED}$$

$$n\langle let\ x_2{:}T_2 = (let\ \mathbf{x_1}{:}\mathbf{T_1} = \mathbf{e_1}\ in\ \mathbf{e})\ in\ t\rangle \rightsquigarrow n\langle let\ \mathbf{x_1}{:}\mathbf{T_1} = \mathbf{e_1}\ in (let\ x_2{:}T_2 = \mathbf{e}\ in\ t)\rangle \qquad \text{LET}$$

$$\mathbf{c}[\![O]\!] \parallel n\langle let\ x{:}T = new\ \mathbf{c}\ in\ t\rangle \rightsquigarrow c[\![O]\!] \parallel \nu\mathbf{o}{:}\mathbf{T}.\ (n\langle let\ x{:}T = \mathbf{o}\ in\ t\rangle \parallel \mathbf{o}[\mathbf{O}].\ ) \qquad \text{NEWO}_i$$

$$\mathbf{o}[O] \parallel n\langle let\ x{:}T = \mathbf{o}.\mathbf{l}(\vec{\mathbf{v}})\ in\ t\rangle \xrightarrow{\tau} o[O] \parallel n\langle let\ x{:}T = \mathbf{O}.\mathbf{l}(\mathbf{o})(\vec{\mathbf{v}})\ in\ t\rangle \qquad \text{CALL}_i$$

Semantics: External steps

- "typed" operational semantics
- i.e., labeled steps between typing judgments:
  $\Delta \vdash P : \Theta$
  - Δ = "assumptions"
    - names assumed present in the rest
  - Θ = "commitments"
    - names guaranteed to the rest
- steps labeled by
  - thread id
  - communicated values
  - kind of communication (!/?, call/return)

External steps (2)

- e.g.: outgoing calls and incoming returns

$$\frac{o \ in \ \Delta}{\Delta \vdash C \parallel n\langle let\, x{:}T = o.l(\vec{v})\ in\ t\rangle : \Theta \xrightarrow{\mathbf{n}\langle\ call\ \mathbf{o.l}(\vec{\mathbf{v}})\rangle!} \Delta \vdash C \parallel n\langle let\, x{:}T = block\ in\ t\rangle : \Theta} \text{ C}$$

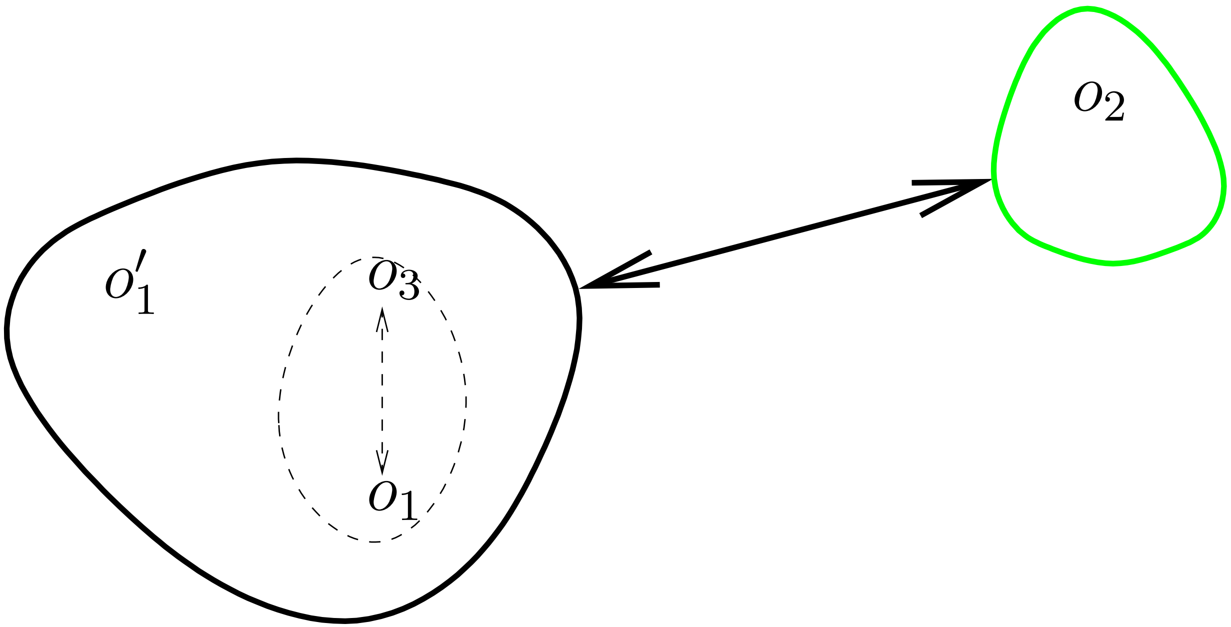
$$\frac{;\Delta, \Theta \vdash v : T}{\Delta \vdash C \parallel n\langle let\, x{:}T = block\ in\ t\rangle : \Theta \xrightarrow{\mathbf{n}\langle\ return(\mathbf{v})\rangle?} \Delta \vdash C \parallel n\langle \mathbf{t}[\mathbf{v}/\mathbf{x}]\rangle : \Theta} \text{ RETURNIN}$$

External steps: Scoping

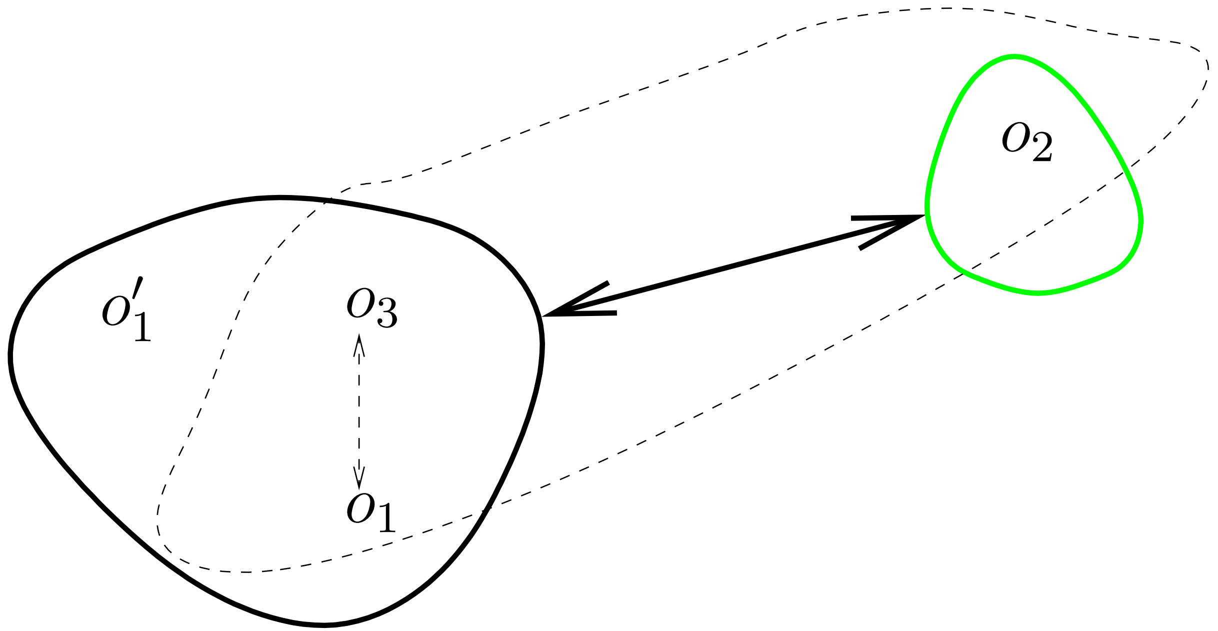
- names
  - for object and thread id's
  - can be generated freshly: "new"
  - valid within dynamic scopes
  - up-to renaming
- dynamic, i.e.,
  - names can be sent around: scope is extended
  - also: across component interface
  - bound exchange of names: "ν"

External steps: Scoping

internal o_3 is sent outside, as argument to method call at o_2



External steps: Scoping



$$\dfrac{\dfrac{o_2 \in \Delta}{\Delta \vdash n\langle o_2.l(o_3); t\rangle : \Theta, \mathbf{o_3{:}T_3} \xrightarrow{n\langle o_1 \ \ call \ \ o_2.l(o_3)\rangle!} \Delta \vdash n\langle block; t\rangle : \Theta, o_3{:}T_3} \text{CALLOUT}}{\Delta \vdash \nu \mathbf{o_3}.(n\langle o_2.l(o_3); t\rangle \parallel o_3[\ldots]) : \Theta \xrightarrow{\nu o_3.\ n\langle o_1 \ \ call \ \ o_2.l(o_3)\rangle!} \Delta \vdash n\langle block; t\rangle \parallel o_3[\ldots] : \Theta, \mathbf{o_3{:}T_3}} \text{OUT}_2$$

External steps: Object creation

- instantiation of a class in the context
- external request for instantiation of component class
- scope of the new id: immediate scope extrusion

$\Rightarrow$ extension of Δ, resp. Θ

$$\frac{c \in \Delta}{\Delta \vdash n\langle let\, x{:}T = new\, c\, in\, t\rangle : \Theta \xrightarrow{\nu\ \mathbf{o_3}{:}\mathbf{T}.creates\ \mathbf{o_3}!} \Delta, \mathbf{o_3}{:}\mathbf{T} \vdash n\langle let\, x{:}T = o_3\, in\, t\rangle : \Theta}\ \textsc{NewO}$$

$$\frac{C(c) = [\![O]\!] \qquad c \in \Theta}{\Delta \vdash C : \Theta \xrightarrow{\nu\ \mathbf{o_3}{:}\mathbf{T}.creates\ \mathbf{o_3}?} \Delta \vdash C \parallel \mathbf{o_3}[\mathbf{O}] : \Theta, \mathbf{o_3}{:}\mathbf{T}}\ \textsc{NewI}$$

F-A in an object-based conc. setting

- [JR02]: for the concurrent ν-calculus
- notion of observation: may-testing equivalence. Formalized here: whether a specific context method ("$o.success()$") is called
- component = set of parallelly running objects + threads
- observable: message exchange at the boundary

$\Rightarrow$ fully abstract observable behavior = communication traces of the labels of the OS

---

actually: they use may-preorder.

What changes?

- classes are the units of exchange: $\mathcal{C}[\mathbf{n}[\![\mathbf{O}]\!]]$!
- i.e., internal and external classes
- component objects can instantiate external classes

can one use these objects for “observations”?

- instances of external classes,
  - instantiation itself is unobservable
  - comm. between component and object observable
  - but:
    - their existence is (principally) unknown to the rest of environment ($\neq$ OC),
    - unless the component gives away their identity!

Consequences/Completeness: Idea

- starting point: component's semantics = set of traces
- Expressibility
- 2 problems for completeness (apart from many technicalities)
  1. expressibility $\Rightarrow$: what are legal traces?
  2. what can be observed/distinguished?

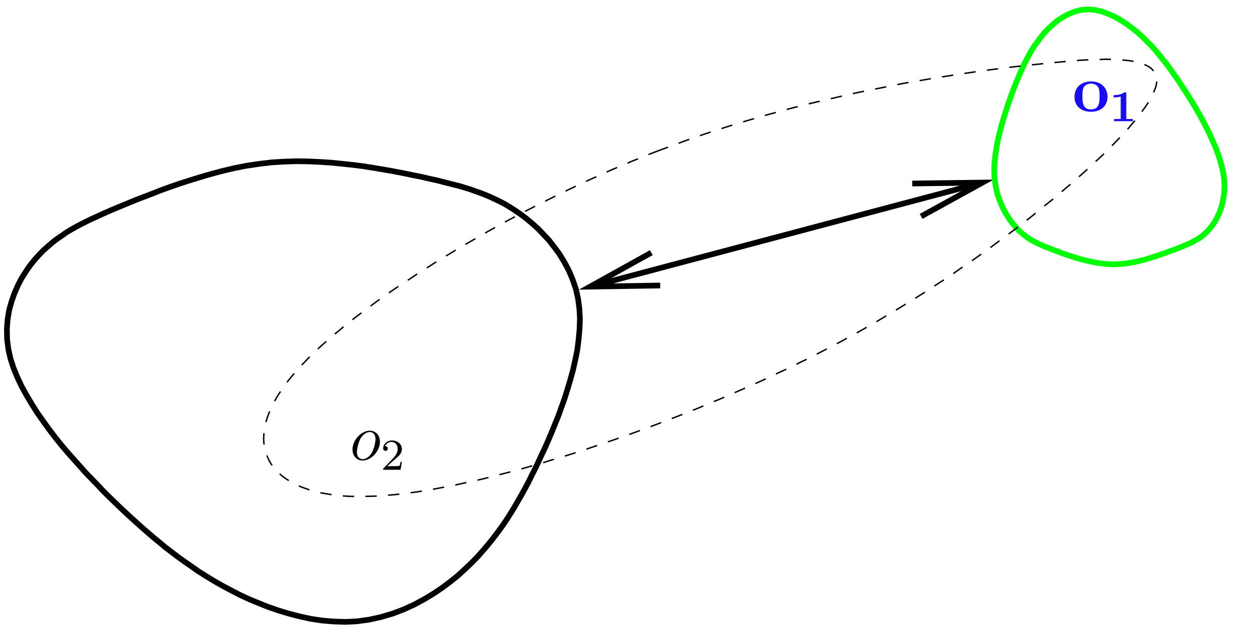
tf*rrr*

Legal traces

- For completeness: component must realize all potential traces but not more!
- various aspects
  - “global”: call-return discipline = balanced/“parenthetic” (per thread)
  - “local”
    - no name clashes: scoping/renaming
    - well-typedness
    - impossible name communication (input)

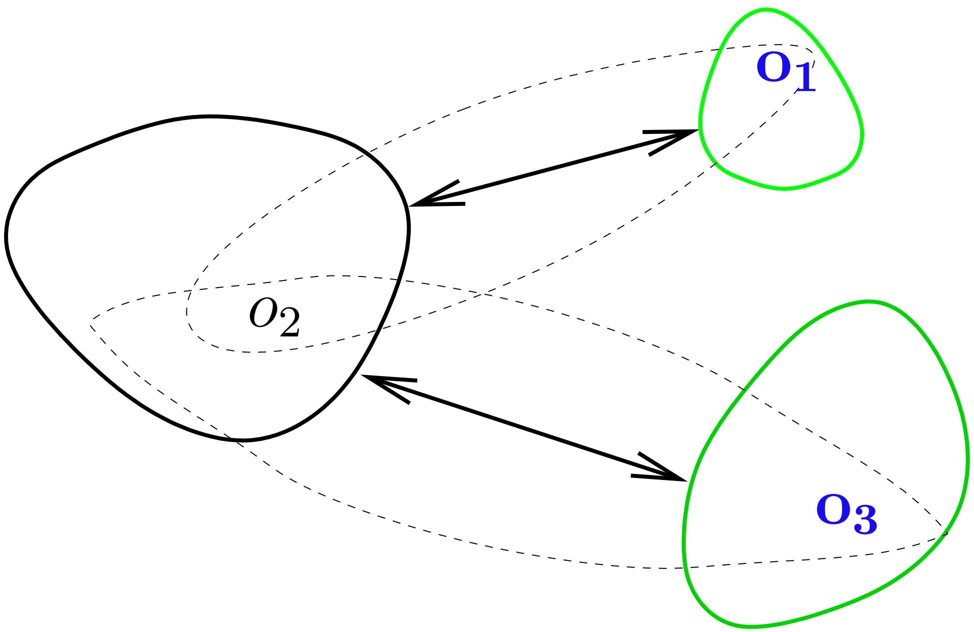
Impossible incoming names?

- Assume: component instantiates two external classes (into o_1 and o_3)



- can o_1 call the component with o_3 as argument?

Impossible incoming names?



- trace labelled

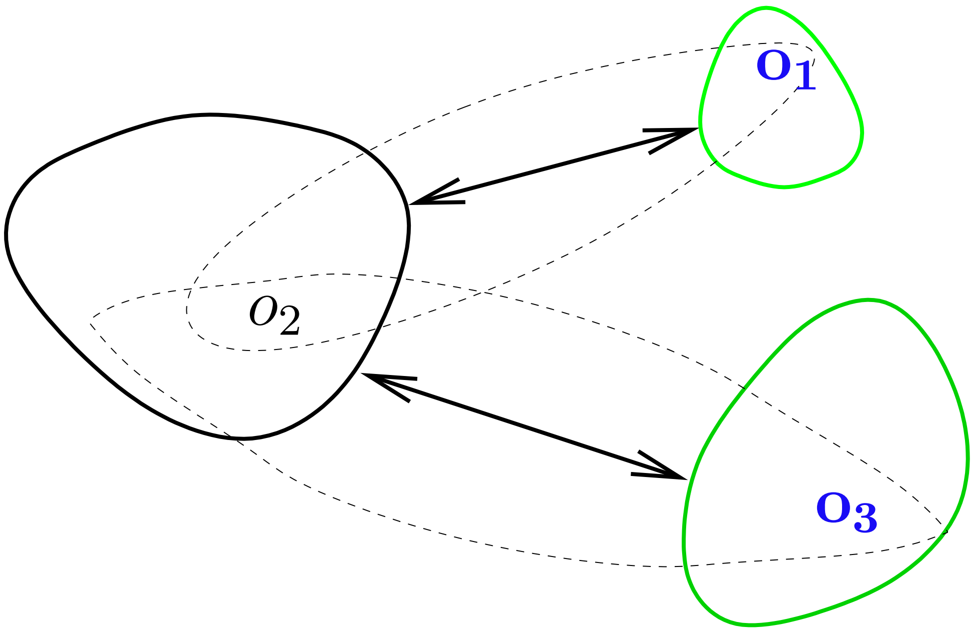
$$\nu o_1.o_2\,creates\,\mathbf{o_1}!.\ \nu o_3.o_2\,creates\,\mathbf{o_3}!.\ n\langle \mathbf{o_1}\ call\ o_2.l(\mathbf{o_3})\rangle?$$

impossible!

What can be distinguished?

- situation as before:

  o_1 and o_3 created externally by component



What can be distinguished?

- instantiation itself is not observable
- communication with the 2 objects is observable
- but!: existence of o_1 unknown to o_3, and vice versa

⇒ observable are communication traces from/to o_1 and from/to o_3

but not their mutual order!

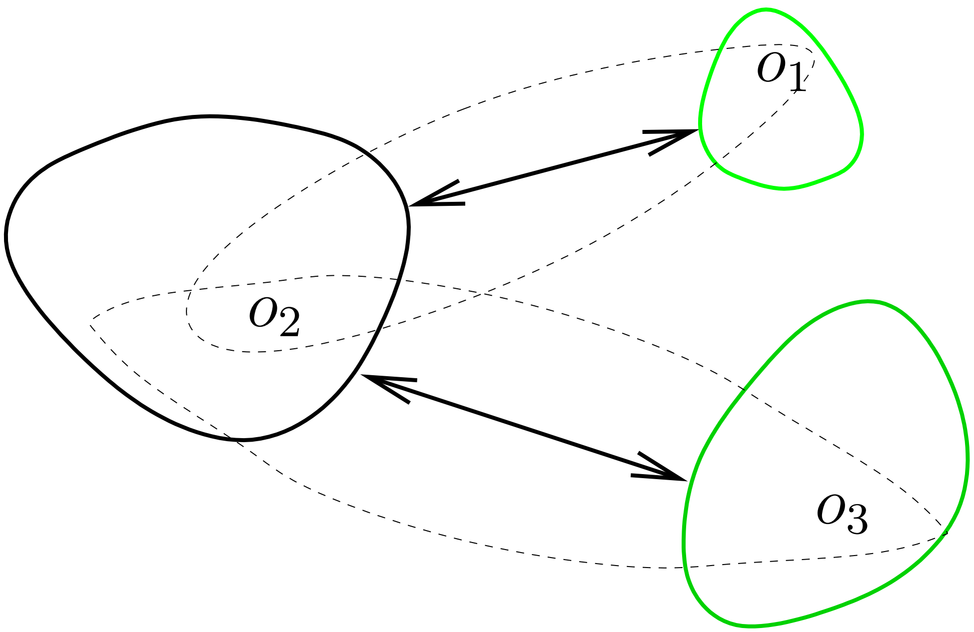
⇒ separated trace sets

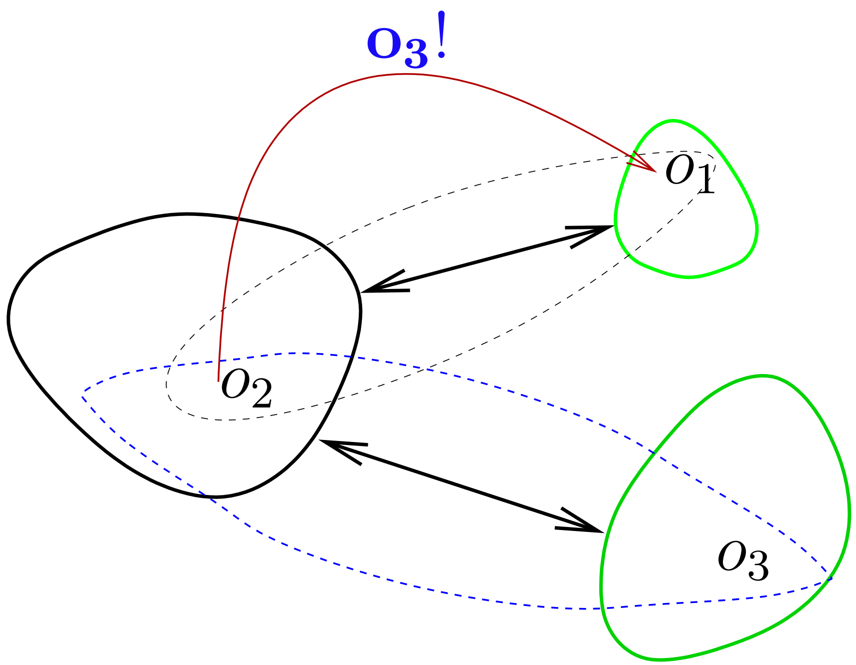
- o_1 and o_3
  - cannot occur in the same label and
  - cannot determine the order of events mutually,

  because

  $\boxed{\text{they don't "know" of each other}}$

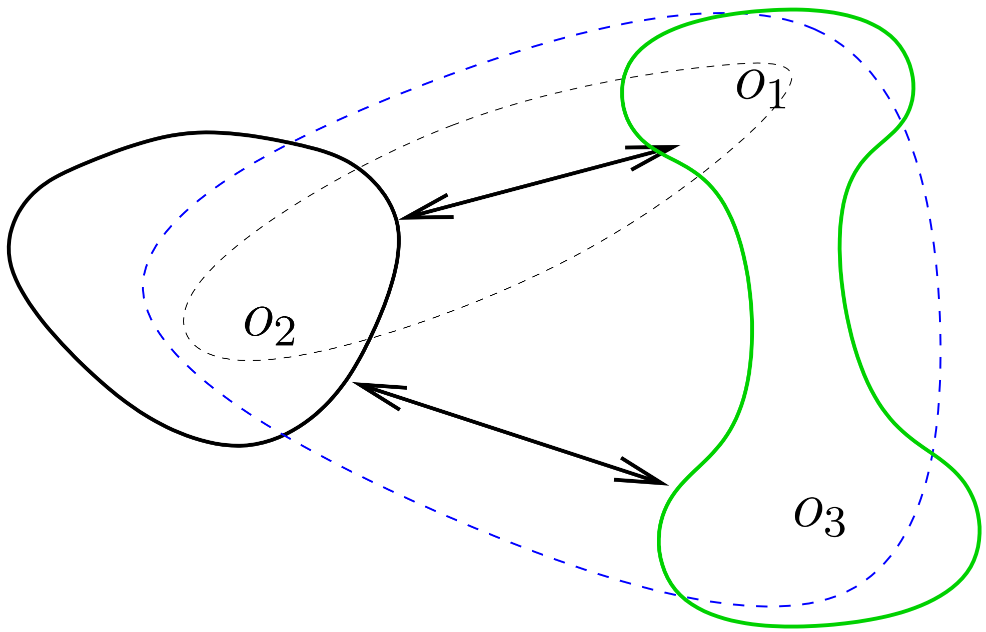
- if "connected", they
  - could occur in the same label and
  - could (in principle) cooperate to observe the order
- connectivity or "acquaintance" is dynamic
- the only one to make o_2 and o_3 acquainted: the component


o_1
o_2
o_3



$$\Delta \vdash n\langle o_1.l(o_3);t\rangle \parallel o_2[\ldots] : \Theta, o_2{:}T_2 \xrightarrow{n\langle o_2\ \mathit{call}\ o_1.l(o_3)\rangle!}$$
$$\Delta \vdash n\langle \mathit{block};t\rangle \parallel o_2[\ldots] : \Theta, o_2{:}T_2$$

- no scope extrusion from perspective of the component



- scope enlarged
- o_1 knows o_3

$\Rightarrow$ o_3 could know now o_1, too

What can be observed, then?

- observers: not just one static outside context but
- dynamic cliques of acquainted objects
  - existing cliques only grow larger: merging
  - new ones can be created by the component
- for full-abstraction:
  - traces per clique, partial-order semantics
  - worst-case: “conspiracy” of environment
- acquaintance = equivalence relation on object id’s

$\Rightarrow$ component keeps track of (the worst-case) of cliques
$\Rightarrow$ set of equations; clique: implied equational theory

- e.g., sending o_1 to o_2, adds $o_1 \hookrightarrow o_2$ to the equations

Approximating the mutual knowledge

- component keeps book about “whom it told what”
- transitions

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

- $E \subseteq \Delta \times (\Delta + \Theta)$ = pairs of objects
- written $\mathbf{o_1} \hookrightarrow \mathbf{o_2}$:
- worst case: equational theory implied by E (on Δ):

$$\mathbf{E} \vdash \mathbf{o_1} \leftrightharpoons \mathbf{o_2}$$

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

Outgoing communication

- outgoing call to o_2, $\Rightarrow$ callee now may know the arguments

$\Rightarrow$ extend E

$$\frac{o_2 \in \Delta \qquad E' = E + (\mathbf{o_2} \hookrightarrow \vec{\mathbf{v}})}{E;\Delta \vdash C \parallel n\langle let\, x{:}T = o_1\ o_2.l(\vec{v})\ in\, t\rangle : \Theta \xrightarrow{n\langle o_1\ \ call\ \ o_2.l(\vec{v})\rangle!} \mathbf{E'};\Delta \vdash C \parallel n\langle let\, x{:}T = block?o_1\ in\, t\rangle : \Theta}$$

- same when an argument is sent ν-bound (where Θ is extended, as well)

Incoming communication

- E unchanged (in first approx.)
- checking for legality:

  is, according to E, the incoming label possible?

$$\frac{;\Delta, n{:}\, \mathit{thread}, \Theta \vdash o_2.l(\vec{v}) : T \qquad o_1 \in \Delta \qquad o_2 \in \Theta \qquad E \vdash \mathbf{o_2} \hookleftarrow;\leftrightharpoons \mathbf{o_1} \qquad E \vdash \mathbf{v} \hookleftarrow;\leftrightharpoons \mathbf{o_1} \vee E \vdash \mathbf{v} \leftrightharpoons \mathbf{o_1}}{E;\Delta, n{:}\, \mathit{thread} \vdash C : \Theta \xrightarrow{n\langle o_1 \ \mathit{call} \ o_2.l(\vec{v})\rangle ?} E;\Delta \vdash C \parallel n\langle \mathit{let}\ x{:}T = \ldots\rangle : n{:}\, \mathit{thread}, \Theta} \ \textsc{Call}$$

Incoming bound values

- incoming ν-bound value

$\Rightarrow$ value new to the component (i.e., not (yet) in Δ)

$$E; \Delta \vdash C : \Theta \xrightarrow{\nu \mathbf{o_3}{:}\mathbf{T_3} \nu o_1{:}T_1.n\langle o_1 \ \ call \ \ o_2.l(o_3)\rangle?}$$

tfrrr

Incoming bound values

- incoming ν-bound value

$\Rightarrow$ value new to the component (i.e., not (yet) in Δ)

- $\Delta' = \Delta, \mathbf{o_3{:}T_3}$

$$\frac{E; \Delta' \vdash C : \Theta \xrightarrow{\nu \mathbf{o_1{:}T_1}.n\langle o_1 \ \ call \ \ o_2.l(o_3)\rangle?}}{E; \Delta \vdash C : \Theta \xrightarrow{\nu o_3{:}T_3 \nu o_1{:}T_1.n\langle o_1 \ \ call \ \ o_2.l(o_3)\rangle?}}$$

Incoming bound values

- incoming ν-bound value

$\Rightarrow$ value new to the component (i.e., not (yet) in Δ)

- $\boldsymbol{\Delta''} = \Delta', \mathbf{o_1{:}T_1}, \mathbf{E''} = E, \mathbf{o_1 \hookrightarrow o_2, o_1 \hookrightarrow o_3}$

$$\dfrac{\dfrac{\dfrac{}{\mathbf{E''}; \boldsymbol{\Delta''} \vdash C : \Theta \xrightarrow{n\langle o_1 \;\; call \;\; o_2.l(o_3)\rangle?}}}{E; \Delta' \vdash C : \Theta \xrightarrow{\nu o_1{:}T_1.n\langle o_1 \;\; call \;\; o_2.l(o_3)\rangle?}}}{E; \Delta \vdash C : \Theta \xrightarrow{\nu o_3{:}T_3 \nu o_1{:}T_1.n\langle o_1 \;\; call \;\; o_2.l(o_3)\rangle?}}$$

Incoming bound values

- incoming ν-bound value

$\Rightarrow$ value new to the component (i.e., not (yet) in Δ)

$$\dfrac{\dfrac{\dfrac{o_2 \in \Theta \qquad o_1 \in \Delta'' \qquad \mathbf{E''} \vdash \mathbf{o_1} \leftrightharpoons \mathbf{o_3} \qquad \mathbf{E''} \vdash \mathbf{o_2} \hookleftarrow ; \leftrightharpoons \mathbf{o_1}}{E''; \Delta'' \vdash C : \Theta \xrightarrow{n\langle o_1 \ \ call \ \ o_2.l(o_3)\rangle?}}}{E; \Delta' \vdash C : \Theta \xrightarrow{\nu o_1{:}T_1.n\langle o_1 \ \ call \ \ o_2.l(o_3)\rangle?}}}{E; \Delta \vdash C : \Theta \xrightarrow{\nu o_3{:}T_3 \nu o_1{:}T_1.n\langle o_1 \ \ call \ \ o_2.l(o_3)\rangle?}}$$

Incoming bound values

- incoming ν-bound value

$\Rightarrow$ value new to the component (i.e., not (yet) in Δ)

$$\dfrac{\dfrac{\dfrac{\begin{array}{cc} o_2 \in \Theta & o_1 \in \Delta'' \\ E'' \vdash o_1 \leftrightharpoons o_3 & E'' \vdash o_2 \hookleftarrow ; \leftrightharpoons o_1 \end{array}}{E''; \Delta'' \vdash C : \Theta \xrightarrow{n\langle o_1 \ \ call \ \ o_2.l(o_3)\rangle?} E''; \Delta'' \vdash C' : \Theta'}}{E; \Delta' \vdash C : \Theta \xrightarrow{\nu o_1{:}T_1.n\langle o_1 \ \ call \ \ o_2.l(o_3)\rangle?} E''; \Delta'' \vdash C' : \Theta'}}{E; \Delta \vdash C : \Theta \xrightarrow{\nu o_3{:}T_3 \nu o_1{:}T_1.n\langle o_1 \ \ call \ \ o_2.l(o_3)\rangle?} E''; \Delta'' \vdash C' : \Theta'}$$

Summary

- in the setting of [JR02] = may-testing equivalence
  - exactly one kind of observation (e.g., “success”)
  - terminal i.e., not repeated observation

$\Rightarrow$ trace semantics gets weakened into a partial order semantics, relative to

  - dynamic cliques of connectivity of objects

- note: we don’t allow to observe (e.g.) divergence!
- note: if we allowed
  - different, repeated observations (for instance success-method + divergence), or
  - if we had a global shared variables (e.g., stdout)

  we are back in linear trace semantics

Status

- operational semantics clear, a generalization of the concurrent ν-calc.
- type system formalized
- exact formulation of the partial-order trace semantics (and proofs $\ldots$)

Conclusions

- are classes good composition units?
- what about cloning?
  - cloning means: obtaining an identical copy (up-to the object identity) of an object “on the run”
  - tree semantics
  - bisimulation equivalence instead of traces
- lock-sychronization
- subtype polymorphism & subclassing
- technology transfer to the proof systems, compositionality

References

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996.

[ÁMdB00] Erika Ábrahám-Mumm and Frank S. de Boer. Proof-outlines for threads in Java. In Catuscia Palamidessi, editor, *Proceedings of CONCUR 2000*, volume 1877 of *Lecture Notes in Computer Science*. Springer-Verlag, August 2000.

[ÁMdBdRS02a] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. A compositional operational semantics for Java$_{MT}$. Technical Report TR-ST-02-2, Lehrstuhl für Software-Technologie, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, May 2002.

[ÁMdBdRS02b] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Verification for Java's reentrant multithreading concept. In Mogens Nielsen and Uffe H. Engberg, editors, *Proceedings of FoSSaCS 2002*, volume 2303 of *Lecture Notes in Computer Science*, pages 4–20. Springer-Verlag, April 2002. A longer version, including the proofs for soundness and completeness, appeared as Technical Report TR-ST-02-1, March 2002.

[GH98] Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. In Uwe Nestmann and Benjamin C. Pierce, editors, *Proceedings of HLCL '98*, volume 16.3 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998.

[JR02] Alan Jeffrey and Julian Rathke. A fully abstract may testing semantics for concurrent objects. In *Proceedings of LICS '02*. IEEE, Computer Society Press, July 2002.

[Mil77] Robin Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, September 1992.

[Plo77] Gordon Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[PS93] A. M. Pitts and D. B. Stark. Observable properties of higher-order functions that dynamically create local names, or: What's new. In Andrzej M. Borzyszkowski and Stefan Sokołowski, editors, *Proceedings of MFCS '93*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, September 1993.

[SW01] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.