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 $\lambda$-calculus/LCF
- various variations of the theme
**Full abstraction: standard setup**

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

  $\Rightarrow$ contextual, observational

- context $C[\_]= \text{“program with a hole”}$
- filling the hole with a *part* of a program (component C): complete program $C[C']$
- what is a *context/component*?: depends on the language/syntax (sequential/parallelfunctional ... contexts)
\begin{itemize}
  \item given a \textbf{closed} program \( P \): \( \mathcal{O}(P) = \text{observations} \)
  
  \Rightarrow \text{observational equivalence:}

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

  \item given a \textbf{denotational} semantics \( \llbracket \_ \rrbracket_D \), resp. the implied equality \( \equiv_D \)
  
  \Rightarrow \equiv_D \text{ is fully abstract wrt. } \equiv_{\text{obs}}:

  \[ \equiv_{\text{obs}} = \equiv_D \]
\end{itemize}
Object calculus: informal

- formal model(s) of oo languages
- in the tradition of the \( \lambda \)-calculi, process calculi . . .
- more specifically:
  - object-calculi of Abadi/Cardelli [AC96]
  - \( \pi \)-calculus: processes, parallelism, name-passing [MPW92][SW01]
  - \( \nu \)-calculus: \( \lambda \)-calc. with name creation (references) respectively its concurrent version [PS93][GH98]
Concurrent $\nu$-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
  - $\nu n : T. (C_1 \parallel C_2)$
  - $\nu$ acts as binder: $\alpha$-equivalence
  - communication of names changes the scope (“scope extrusion”)

- methods = functions with specific “self”-parameter  

- active entities: threads
  - sequencing + local, static scoping: $let \ x = \ e \ in \ t$
  - thread creation

---

\[ ^a \text{In the presence of subtyping, the parameter would be late-bound.} \]
Concurrent $\nu$-calculus: Syntax

\[
P ::= \mathbf{0} \mid P \parallel P \mid \nu(n:T).P \mid n[O] \mid n\langle t \rangle \text{ components}
\]

\[
O ::= l = m, \ldots, l = m \text{ object}
\]

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

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

\[
e ::= t \mid \text{if } v = v \text{ then } e \text{ else } e \text{ expressions}
\]

\[
| \quad v.l(v, \ldots, v) \mid n.l \leftarrow m
\]

\[
| \quad \text{new}[O] \mid \text{new}\langle t \rangle \mid \text{currentthread}
\]

\[
v ::= x \mid n \text{ values}
\]
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
    - \( \nu \)-binding (= hiding)
    - storing, sending, receiving etc.

- method update not used in class-based setting
Adding classes

\[ P ::= 0 | n[(O)] | n\langle t \rangle | P \parallel P | R \quad \text{program (stat.)} \]
\[ R ::= 0 | R \parallel R | \nu(n:T).R | n[O] | n\langle t \rangle \quad \text{program (dyn.)} \]
\[ O ::= l = m, \ldots , l = m \quad \text{object} \]
\[ m ::= \varsigma(n:T).\lambda(x:T, \ldots , x:T).t \quad \text{method} \]
\[ t ::= v | \text{stop} | \text{let } x:T = e \text{ in } t \quad \text{thread} \]
\[ e ::= t | \text{if } v = v \text{ then } e \text{ else } e | v.l(v, \ldots , v) \quad \text{expression} \]
\[ e ::= \text{new } n \mid \text{new } n\langle t \rangle \]
\[ v ::= x \mid n \quad \text{values} \]
Semantics (1)

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

- black: objects of the component
- green: objects of the environment
• $o_1$ creates an internal object $o_3$ (assume: thread $n$ visits $o_1$)

\[
\begin{align*}
\text{c}[O] & \parallel n \langle \text{let } x : T = \text{new c in } t \rangle \rightsquigarrow \\
\text{c}[O] & \parallel \nu o_3 : T. (n \langle \text{let } x : T = o_3 \text{ in } t \rangle \parallel o_3[O]. )
\end{align*}
\]
Semantics: Internal steps

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

\[
\begin{align*}
\text{RED} & \quad n\langle \text{let } x:T = v \text{ in } t \rangle \rightsquigarrow n\langle t[v/x] \rangle \\
\text{LET} & \quad n\langle \text{let } x_2:T_2 = (\text{let } x_1:T_1 = e_1 \text{ in } \ e) \text{ in } t \rangle \rightsquigarrow n\langle \text{let } x_1:T_1 = e_1 \text{ in } (\text{let } x_2:T_2 = e \text{ in } t) \rangle \\
\text{NEWO}_i & \quad c[O] \parallel n\langle \text{let } x:T = \text{new } c \text{ in } t \rangle \rightsquigarrow c[O] \parallel \nu o:T. \ (n\langle \text{let } x:T = o \text{ in } t \rangle \parallel o[O].) \\
\text{CALL}_i & \quad o[O] \parallel n\langle \text{let } x:T = o.l(\vec{v}) \text{ in } t \rangle \rightarrow o[O] \parallel n\langle \text{let } x:T = O.l(o)(\vec{v}) \text{ in } t \rangle
\end{align*}
\]
Semantics: External steps

- “typed” operational semantics
- i.e., labeled steps between typing judgments: \( \Delta \vdash P : \Theta \)
  - \( \Delta = \text{“assumptions”} \)
    - names assumed present in the rest
  - \( \Theta = \text{“commitments”} \)
    - 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

\[
\begin{align*}
\Delta \vdash C \parallel n\langle \text{let } x : T = o.l(\vec{v}) \text{ in } t \rangle &: \Theta^n \frac{\text{call } o.l(\vec{v})}{\Delta \vdash C \parallel n\langle \text{let } x : T = \text{block in } t \rangle &: \Theta} \\
&; \Delta, \Theta \vdash v : T \\
\Delta \vdash C \parallel n\langle \text{let } x : T = \text{block in } t \rangle &: \Theta^n \frac{\text{return}(v)}{\Delta \vdash C \parallel n\langle t[v/x] \rangle &: \Theta}
\end{align*}
\]
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 $o_3$ is sent outside, as argument to method call at $o_2$
$\Delta \vdash n\langle o_2.l(o_3); t \rangle : \Theta, o_3 : T_3$ 

$\Delta \vdash n\langle o_2.l(o_3); t \rangle : \Theta, o_3 : T_3$ 

$\Delta \vdash \nu o_3.\langle n\langle o_2.l(o_3); t \rangle \parallel o_3[\ldots] \rangle : \Theta$ 

$\Delta \vdash \nu o_3.\langle n\langle o_2.l(o_3); t \rangle \parallel o_3[\ldots] \rangle : \Theta, o_3 : T_3$ 

$\Delta \vdash n\langle block; t \parallel o_3[\ldots] \rangle : \Theta, o_3 : T_3$
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
  ⇒ extension of $\Delta$, resp. $\Theta$

\[
\frac{c \in \Delta}{\Delta \vdash n\langle \text{let } x:T = \text{new } c \text{ in } t \rangle : \Theta} \quad \text{NEWO}
\]

\[
\frac{\Delta \vdash n\langle \text{let } x:T = o_3 \text{ in } t \rangle : \Theta}{\Delta, o_3:T \vdash n\langle \text{let } x:T = o_3 \text{ in } t \rangle : \Theta}
\]

\[
\frac{C(c) = [O]}{\Delta \vdash C : \Theta} \quad \text{NEWL}
\]

\[
\Delta \vdash C : \Theta \quad \frac{\nu o_3:T.create \ o_3}{\Delta \vdash C \parallel o_3[O] : \Theta, o_3:T}
\]
F-A in an object-based conc. setting

- [JR02]: for the concurrent $\nu$-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: \( C[n(O)]! \)
- 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?
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/re naming
    - 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

\[ \nu_1.o_2 creates o_1!. \nu_3.o_2 creates o_3!. n\langle o_1 \text{ call } o_2.l(o_3) \rangle? \]

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
  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
\[ \Delta \vdash n\langle o_1.l(o_3); t \rangle \parallel o_2[\ldots] : \Theta, o_2:T_2 \]

\[ \Delta \vdash n\langle block; t \rangle \parallel o_2[\ldots] : \Theta, o_2:T_2 \]

- 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
  ⇒ component keeps track of (the worst-case) of cliques
  ⇒ set of equations; clique: implied equational theory
- e.g., sending $o_1$ to $o_2$, adds $o_1 \leftrightarrow o_2$ to the equations
Approximating the mutual knowledge

- component keeps book about “whom it told what”
- transitions
  \[ E; \Delta' \vdash C : \Theta \xrightarrow{a} E'; \Delta' \vdash C' : \Theta' \]
- \( E \subseteq \Delta \times (\Delta + \Theta) \) = pairs of objects
- written \( o_1 \leftrightarrow o_2 \):
- worst case: equational theory implied by \( E \) (on \( \Delta \)):
  \[ E \vdash o_1 \leftrightarrow o_2 \]
  (for \( o_2 \in \Theta: E \vdash o_1 \leftrightarrow ; \leftrightarrow o_2 \))
Outgoing communication

• outgoing call to \(o_2\), \(\Rightarrow\) callee now may know the arguments

\[ \Rightarrow \text{extend } E \]

\[
\begin{align*}
o_2 \in \Delta & \quad E' = E + (o_2 \leftrightarrow \vec{v}) \\
E; \Delta \vdash C \parallel n(let x:T = o_1 o_2.l(\vec{v}) \ in \ t) : \Theta & \xrightarrow{n(o_1 \ \text{call} \ o_2.l(\vec{v}))} E'; \Delta \vdash C \parallel n(let x:T = \text{block}?o_1 \ in \ t) : \Theta
\end{align*}
\]

• same when an argument is sent \(\nu\)-bound (where \(\Theta\) is extended, as well)
Incoming communication

- $E$ unchanged (in first approx.)
- checking for **legality**: is, according to $E$, the incoming label possible?

\[
\begin{align*}
; \Delta, n: \text{thread}, \Theta \vdash o_2.l(\bar{v}) : T & \quad o_1 \in \Delta \quad o_2 \in \Theta \\
E \vdash o_2 \leftarrow; \Leftarrow o_1 & \quad E \vdash v \leftarrow; \Leftarrow o_1 \lor E \vdash v \Leftarrow o_1
\end{align*}
\]

\[
E; \Delta, n: \text{thread} \vdash C : \Theta \xrightarrow{n\langle o_1 \text{ call } o_2.l(\bar{v})\rangle?} E; \Delta \vdash C \parallel n\langle \text{let } x:T = \ldots \rangle : n: \text{thread}, \Theta
\]
Incoming bound values

- incoming $\nu$-bound value

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

\[ E; \Delta \vdash C : \Theta \quad \frac{\nu_3 : T_3 \nu_1 : T_1.n(o_1 \text{ call } o_2.l(o_3))}{\Rightarrow} \]
Incoming bound values

• incoming $\nu$-bound value
  $\Rightarrow$ value *new* to the component (i.e., not (yet) in $\Delta$)
• $\Delta' = \Delta, o_3 : T_3$

\[
\begin{align*}
\text{if } & E; \Delta' \vdash C : \Theta \quad \nu_1 : T_1. n\langle o_1 \text{ call } o_2. l(o_3) \rangle? \\
\text{then } & E; \Delta \vdash C : \Theta \quad \nu_3 : T_3\nu_1 : T_1. n\langle o_1 \text{ call } o_2. l(o_3) \rangle?
\end{align*}
\]
**Incoming bound values**

- incoming \( \nu \)-bound value
  
  \[ \Rightarrow \text{value new to the component (i.e., not (yet) in } \Delta) \]

- \( \Delta'' = \Delta', o_1: T_1, E'' = E, o_1 \leftarrow o_2, o_1 \leftarrow o_3 \)

\[ E''; \Delta'' \vdash C : \Theta \quad \frac{n\langle o_1 \text{ call } o_2.l(o_3)\rangle?}{\text{sound}} \]

\[ E; \Delta' \vdash C : \Theta \quad \frac{\nu o_1:T_1.n\langle o_1 \text{ call } o_2.l(o_3)\rangle?}{\text{sound}} \]

\[ E; \Delta \vdash C : \Theta \quad \frac{\nu o_3:T_3\nu o_1:T_1.n\langle o_1 \text{ call } o_2.l(o_3)\rangle?}{\text{sound}} \]
Incoming bound values

- incoming \( \nu \)-bound value

\[ \Rightarrow \text{value new to the component (i.e., not (yet) in } \Delta \)\]

\[
\begin{array}{c}
o_2 \in \Theta \\
E'' \vdash o_1 \iff o_3 \\
E'' \vdash o_2 \leftarrow; \iff o_1
\end{array}
\]

\[\begin{array}{c}
E''; \Delta'' \vdash C : \Theta \\
\frac{n\langle o_1 \text{ call } o_2.l(o_3) \rangle?}{n\langle o_1 \text{ call } o_2.l(o_3) \rangle?} \\
E; \Delta' \vdash C : \Theta \\
\frac{\nu o_1: T_1.n\langle o_1 \text{ call } o_2.l(o_3) \rangle?}{\nu o_3:T_3\nu o_1:T_1.n\langle o_1 \text{ call } o_2.l(o_3) \rangle?}
\end{array}\]

\[E; \Delta \vdash C : \Theta \]

\[\frac{\nu o_3:T_3\nu o_1:T_1.n\langle o_1 \text{ call } o_2.l(o_3) \rangle?}{\nu o_3:T_3\nu o_1:T_1.n\langle o_1 \text{ call } o_2.l(o_3) \rangle?}\]
Incoming bound values

- incoming $\nu$-bound value

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

$$
\begin{align*}
    o_2 & \in \Theta & o_1 & \in \Delta'' \\
    E'' \vdash o_1 \iff o_3 & \quad E'' \vdash o_2 \leftarrow; \iff o_1 \\
    E''; \Delta'' \vdash C : \Theta & \quad n\langle o_1 \text{ call } o_2.l(o_3) \rangle? \\
    & \quad E''; \Delta'' \vdash C' : \Theta'
\end{align*}
$$

$$
\begin{align*}
    E; \Delta' \vdash C : \Theta & \quad \nu o_1 : T_1.n\langle o_1 \text{ call } o_2.l(o_3) \rangle? \\
    & \quad E''; \Delta'' \vdash C' : \Theta'
\end{align*}
$$

$$
\begin{align*}
    E; \Delta \vdash C : \Theta & \quad \nu o_3 : T_3\nu o_1 : T_1.n\langle o_1 \text{ call } o_2.l(o_3) \rangle? \\
    & \quad E''; \Delta'' \vdash C' : \Theta'
\end{align*}
$$
Summary

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

⇒ 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
• operational semantics clear, a generalization of the concurrent $\nu$-calc.
• type system formalized
• exact formulation of the partial-order trace semantics (and proofs ... )
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-synchronization
• subtype polymorphism & subclassing
• technology transfer to the proof systems, compositionality
References


