Object connectivity and full abstraction for class-based, multithreaded OO

MobiJ workshop@FMCO, 3rd November, 2003

Erika Ábrahám   Marcello Bonsangue   Frank S. de Boer   Martin Steffen

Overview

- full-abstraction
- class-based calculus
- issues for full abstraction
- completeness and legal traces
- conclusion

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 [10], Plotkin [13] for λ-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 $\mathcal{C}[\_]$ = “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 [1]
  - π-calculus: processes, parallelism, name-passing [11][14]
  - ν-calculus: λ-calc. with name creation (references) respectively its concurrent version [12][8]

Concurrent ν-calculus with classes

- program = "set" of named threads, objects, and classes: $n\langle t \rangle$, $n[c]$ and $n[\![(l_1 = m_1, \ldots, l_k = m_k)]\!]$
- dynamic scoping of names
  - $\nu n{:}T.\ (C_1 \parallel C_2)$
  - communication of names changes the scope ("scope extrusion")
- class = "like" an object that accepts only a new-method; class names are not first-class citizens
- methods = functions with specific "self"-parameter [a]
- active entities: threads
  - sequencing + local, static scoping: $let\, x = e\, in\, t$
  - thread creation

Concurrent ν-calculus with classes

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

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

F-A in an object-based conc. setting

- [9]: 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 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!

Completeness: line of argument

- goal: if $C_1 \equiv_{obs} C_2$, then $C_1 \equiv_{\mathcal{D}} C_2$
- so, given a legal trace $s \in [\![C_1]\!]_{\mathcal{D}}$, do
  - construct a complementary context $\mathcal{C}_{\bar{s}}$
  - composition: program + context do the observation

$$\mathcal{C}_{\bar{s}}[C_1] \longrightarrow^* success$$

  - observational equivalence: C_2 can do that, too:

$$\mathcal{C}_{\bar{s}}[C_2] \longrightarrow^* success$$

  - decomposition: $s \in [\![C_2]\!]_{\mathcal{D}}$

That s is a trace of C_2 by decomposition is not a direct consequence. I ignore that here.

Legal traces

- core of completeness: definability $\Rightarrow$
- for each legal trace s: construct a component C_s realizing it
- first: characterize the legal traces exactly
- derivability of legal-trace-judgement:

$$\Delta; E_\Delta \vdash r \rhd \mathbf{s} : trace\, \Theta; E_\Theta$$

Legal traces: incoming call

- General setup: scan the trace, where
  - r: history
  - as future with next label a

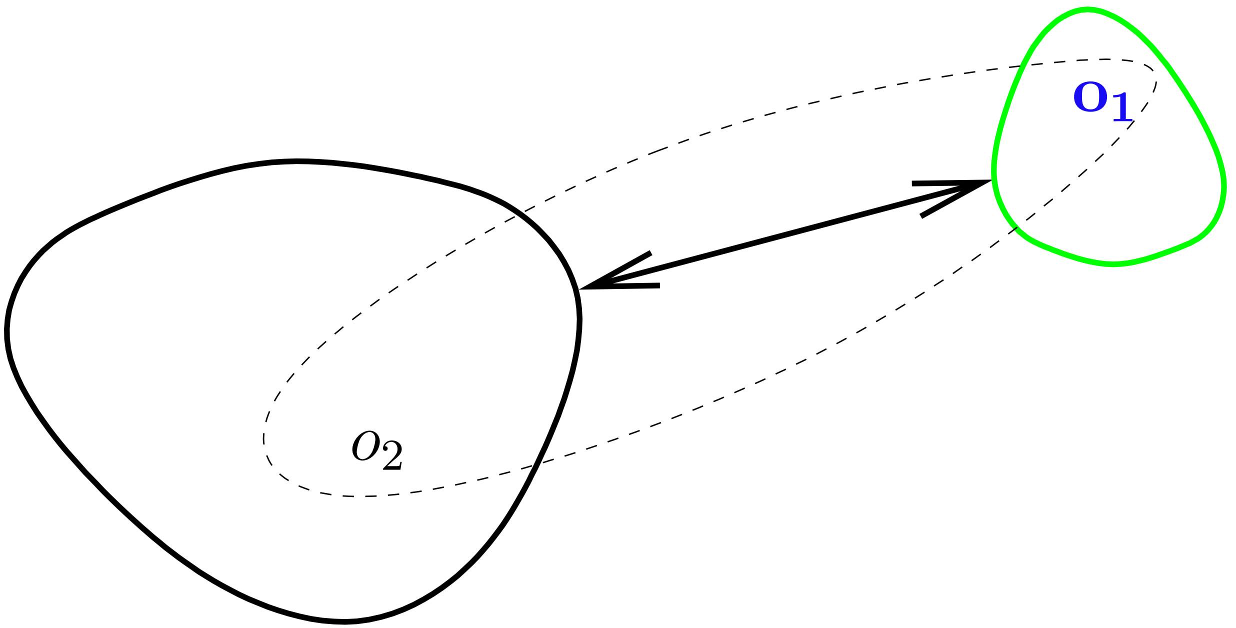
$$\frac{\text{lots of conditions} \quad \acute{\Delta}; \acute{E}_{\Delta} \vdash \mathbf{r\ a} \rhd \mathbf{s} : trace\ \acute{\Theta}; \acute{E}_{\Theta}}{\Delta; E_{\Delta} \vdash \mathbf{r} \rhd \mathbf{a\ s} : trace\ \Theta; E_{\Theta}}$$

“Lots of conditions”

- For completeness: component must realize all possible 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 (“connectivity”)

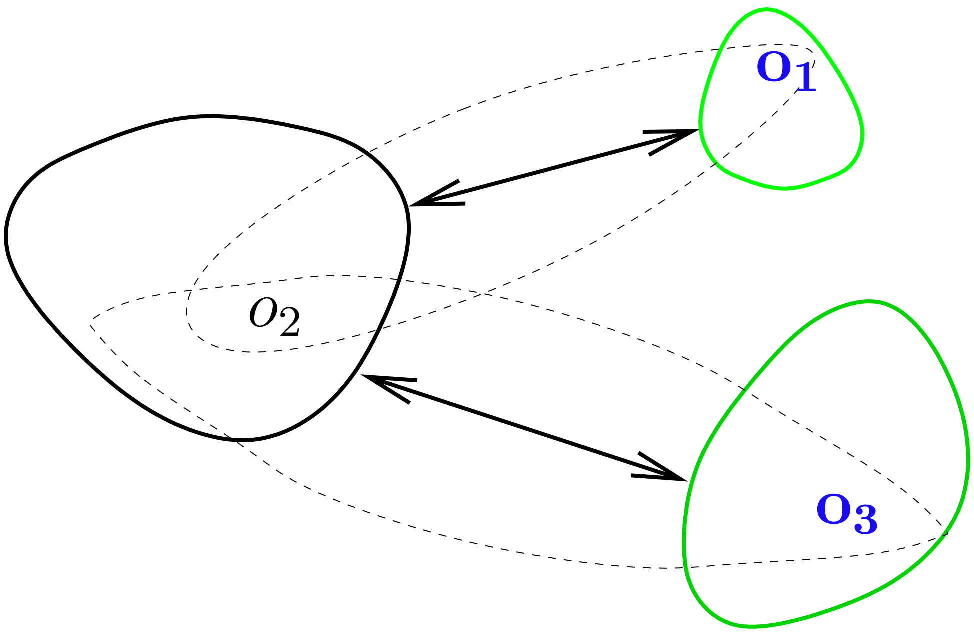
Impossible incoming names?

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



- can o_1 and o_3 be sent in the same argument list? (for example)

Impossible incoming names?



- trace labelled

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

impossible!

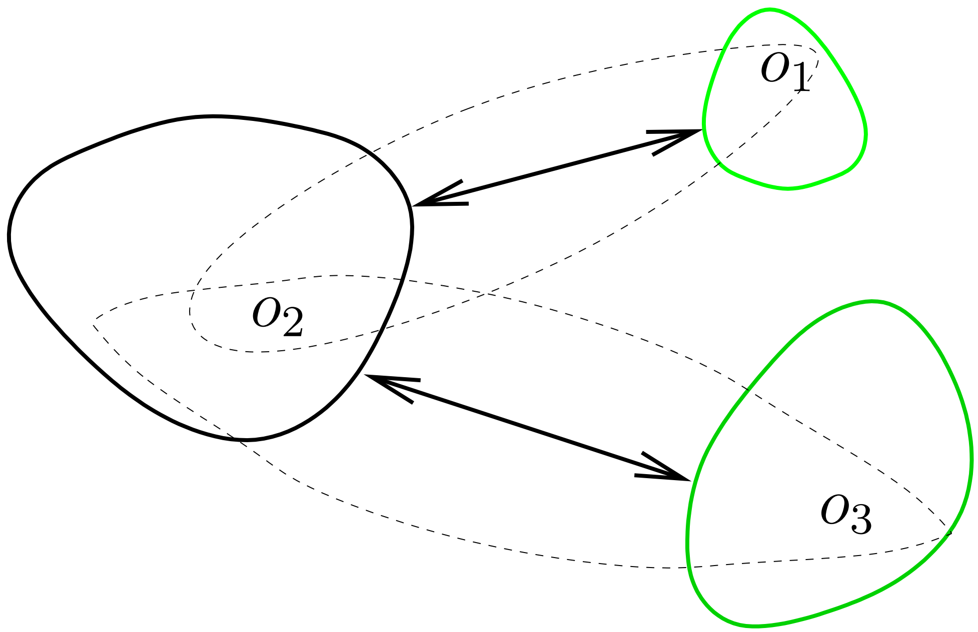
Acquaintance

- o_1 and o_3: cannot occur in the same label and because

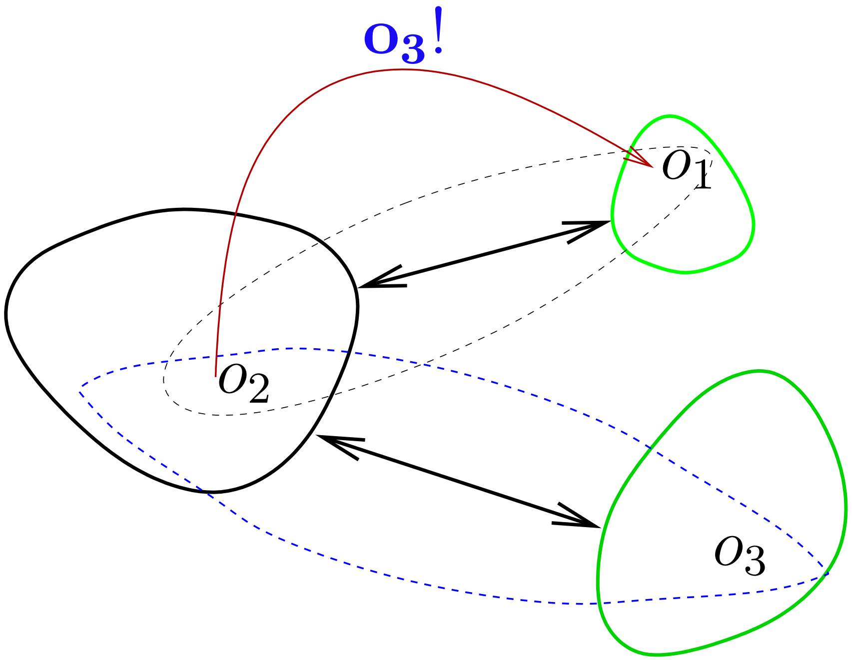
  they do not possibly "know" of each other

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

Dynamic acquaintance



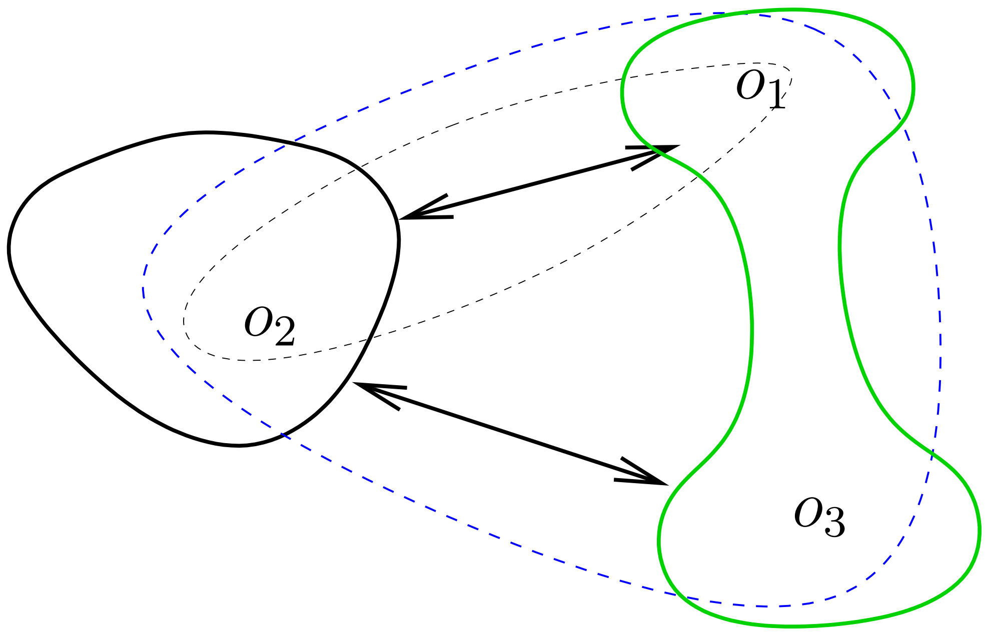
Dynamic acquaintance



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

- no scope extrusion from perspective of the component

Dynamic acquaintance



- scope enlarged
- o_1 knows o_3

$\Rightarrow$ o_3 could know now o_1, too

- and all objects that o_3 knows, could know o_1 in turn, too

. . .

Acquaintance: assumptions and commitments

- acquaintance = equivalence relation on object id's

$\Rightarrow$ keep track of (the worst-case) of connectivity

$\Rightarrow$ set of "equations"; clique: implied equational theory

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

Incoming call: acquaintance

- let $\quad a = n\langle[o_1]call\ o_2.l(\vec{v})\rangle?$

$$\acute{E}_\Theta = E_\Theta + (\mathbf{o_2} \hookrightarrow \vec{\mathbf{v}})$$

$$\frac{E_\Delta \vdash \mathbf{o_1} \leftrightharpoons; \hookrightarrow \vec{\mathbf{v}} \qquad E_\Delta \vdash \mathbf{o_1} \leftrightharpoons; \hookrightarrow \mathbf{o_2} \qquad \Delta; \acute{E}_\Delta \vdash r\ a \rhd s : trace\ \Theta; \acute{\mathbf{E}}_\Theta}{\Delta; E_\Delta \vdash r \rhd a\ s : trace\ \Theta; E_\Theta}$$

Incoming bound value

- bound input: E_Δ extended to $\acute{E}_\Delta$
- crucial question

  What is the connectivity of the new objects?

- we have to guess!

$\Rightarrow$ extend E_Δ to $\acute{E}_\Delta$:

Incoming bound value: arbitrary guess?

- can the extension from E_Δ to E'_Δ be arbitrary?
- No:

> "No news about old objects"

- i.e.,

> "theory of E'_Δ: a conservative extension of E_Δ"

- written: $\mathbf{E}_\Delta \vdash \mathbf{E}'_\Delta \downarrow_{\Delta\times(\Delta+\Theta)}$

Incoming call: bound input

- let $a = \nu(\Delta').\, n\langle[o_1]\mathit{call}\ o_2.l(\vec{v})\rangle?$

$$\frac{\acute{E}_\Theta = E_\Theta + (o_2 \hookrightarrow \vec{v}) \quad \acute{E}_\Delta \vdash o_1 \leftrightharpoons; \hookrightarrow \vec{v} \quad \acute{E}_\Delta \vdash o_1 \leftrightharpoons; \hookrightarrow o_2 \qquad (\acute{\Delta}, \acute{E}_\Delta) = (\Delta, E_\Delta) + \Delta' \quad E_\Delta \vdash \acute{E}_\Delta \downarrow_{\Delta \times (\Delta + \Theta)} \quad \acute{\Delta}; \acute{E}_\Delta \vdash r\ a \rhd s : \mathit{trace}\ \acute{\Theta}; \acute{E}_\Theta}{\Delta; E_\Delta \vdash r \rhd a\ s : \mathit{trace}\ \Theta; E_\Theta}$$

- extend the assumption contexts
- check for conservativity of the guess

One has also to extend the commitments; I omit this here.

Legal traces: balance

- incoming call
- check for input enabledness per thread
- consult the history
- for instance: incoming return a possible in a next step

$$\frac{pop\; n\; r = \nu(\Theta').\; n\langle [o_1]\, call\;\, o_2.l(\vec{v})\rangle !}{\Delta \vdash r \rhd \nu(\Delta').\; n\langle return(v)\rangle ? : \Theta}$$

- before a return: there must have been an outgoing call
- pop picks out the last “matching” call

Incoming comm.: the full story

$$\frac{\begin{array}{c} a = \nu(\Delta', \Theta').\ n\langle[o_1]call\ o_2.l(\vec{v})\rangle? \qquad \acute{E}_\Theta = E_\Theta + (o_2 \hookrightarrow \vec{v}, n \hookrightarrow o_2) \\ (\acute{\Delta}, \acute{E}_\Delta) = (\Delta, E_\Delta) + \Delta' \qquad \Delta; E_\Delta \vdash \acute{E}_\Delta \downarrow_{\Delta\times(\Delta+\Theta)} : \Theta \qquad \acute{\Theta} = \Theta + \Theta' \\ ;\Theta \vdash o_2 : c_2 \quad ;\Theta \vdash c_2 : [\![(\ldots, l{:}\vec{T} \to T, \ldots)]\!] \quad [\acute{\Delta}] \vdash [o_1 : [\ldots]] \quad \acute{\Delta}, \Theta \vdash n{:}\ thread \quad ;\acute{\Delta}, \acute{\Theta} \vdash \vec{v} : \vec{T} \\ dom(\Delta', \Theta') \subseteq fn(n\langle[o_1]call\ o_2.l(\vec{v})\rangle) \\ \acute{\Delta}; \acute{E}_\Delta \vdash [o_1] \leftrightharpoons\!\hookrightarrow \vec{v} : \acute{\Theta} \qquad \acute{\Delta}; \acute{E}_\Delta \vdash [o_1] \leftrightharpoons\!\hookrightarrow o_2 : \acute{\Theta} \qquad \acute{\Delta}; \acute{E}_\Delta \vdash n \leftrightharpoons [o_1] : \acute{\Theta} \\ \Delta \vdash r \rhd a : \Theta \qquad \acute{\Delta}; \acute{E}_\Delta \setminus n \vdash r\ a \rhd s : trace\ \acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash r \rhd a\ s : trace\ \Theta; E_\Theta}$$

Definability

- given a legal trace s $\Rightarrow$ define C_s by

  induction on the derivation for
  $$\Delta; E_\Delta \vdash r \rhd \mathbf{s} : trace\, \Theta; E_\Theta$$

$\Rightarrow$ construct the program backwards!

actions on the commitment context E_Θ:

E_Θ: each object knows its clique, kept up-to date

- giving away new id's: create them

  propagate/broadcast information through the clique

- incoming calls: wrap up the method body, put it into the class

Definability

- for example outgoing call $a = \nu(\Theta').\ n\langle [o_1] call\ o_2.l(\vec{v})\rangle!$
- we know: afterwards

$$\acute{C}_s = n\langle let\, x{:}T = [o_1]\ blocks\ for\ o_2\ in\, t'\rangle \parallel C'_s$$

- construct component $\grave{C}_s$ before the call:

$${}_s = C'_s \parallel n\langle create(\Theta'); propagate(\Theta'); wait(o_2, \vec{v}); o_2.delegate_l(o_1, \vec{v}); t\rangle$$

where $t = let\, x{:}T = [o_1]\ blocks\ for\ o_2\ in\, t'$.

What I didn't mention

- static typing
- treatment of "cross-border" instantiation:
  - instantiation itself is not visible
  - "lazy instantiation"
  - guessing connectivity also for instances the "other side" instantiated in the component (and vice versa)
- caller identity must ultimately be ignored
- coding issues
- objects not acquainted cannot determine relative order of events of each other

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

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.

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

[3] E. Ábrahám, M. M. Bonsangue, F. S. de Boer, and M. Steffen. A structural operational semantics for a concurrent class calculus. Technical Report 0307, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Aug. 2003.

[4] E. Ábrahám, F. S. de Boer, W.-P. de Roever, and M. Steffen. A compositional operational semantics for Java$_{MT}$. In N. Derschowitz, editor, *International Symposium on Verification (Theory and Practice)*, volume 2772 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003. To appear. A preliminary version appeared as Technical Report TR-ST-02-2, May 2002.

[5] E. Ábrahám, F. S. de Boer, W.-P. de Roever, and M. Steffen. A Hoare logic for monitors in Java. Techical report TR-ST-03-1, Lehrstuhl für Software-Technologie, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Apr. 2003.

[6] E. Ábrahám-Mumm and F. S. de Boer. Proof-outlines for threads in Java. In C. Palamidessi, editor, *Proceedings of CONCUR 2000*, volume 1877 of *Lecture Notes in Computer Science*. Springer-Verlag, Aug. 2000.

[7] E. Ábrahám-Mumm, F. S. de Boer, W.-P. de Roever, and M. Steffen. Verification for Java’s reentrant multithreading concept. In M. Nielsen and U. H. Engberg, editors, *Proceedings of FoSSaCS 2002*, volume 2303 of *Lecture Notes in Computer Science*, pages 4–20. Springer-Verlag, Apr.

2002. A longer version, including the proofs for soundness and completeness, appeared as Technical Report TR-ST-02-1, March 2002.

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

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

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

[11] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, Sept. 1992.

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

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

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