Observability, Classes, and Object Connectivity

Erika Ábrahám  Marcello Bonsangue  Frank S. de Boer  Martin Steffen

Christian-Albrechts University Kiel

Summer Research Institute, EPFL Lausanne
July 2004
Structure

Introduction

Calculus

Classes and observable behavior

Object connectivity

Conclusion
Introduction

Calculus

Classes and observable behavior

Object connectivity

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 [4], Plotkin [7] for \(\lambda\)-calculus/LCF
- various variations of the theme
Full abstraction: standard setup

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

  =>

- 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/parallel/functional \ldots contexts)
F-A: standard setup (cont’d)

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

\[
C_1 \equiv_{\text{obs}} C_2 \iff \forall C. \mathcal{O}(C[C_1]) = \mathcal{O}(C[C_2])
\]

- given a denotational semantics $\llbracket \cdot \rrbracket_D$, resp. the implied equality $\equiv_D$
  $\Rightarrow \equiv_D$ is fully abstract wrt. $\equiv_{\text{obs}}$:

\[
\equiv_{\text{obs}} = \equiv_D
\]
Introduction

Calculus

Classes and observable behavior

Object connectivity

Conclusion
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 [1]
  - $\pi$-calculus: processes, parallelism, name-passing [5][8]
  - $\nu$-calculus: $\lambda$-calc. with name creation (references) respectively its concurrent version [6][2]
Concurrent $\nu$-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, i.e. not subject to
  - storing, sending, receiving etc.
- methods = functions with specific “self”-parameter\(^1\)
- active entities: threads
  - sequencing + local, static scoping: $let \ x = e \ in \ t$
  - thread creation

\(^1\)In the presence of subtyping, the parameter would be late-bound.
Concurrent $\nu$-calculus with classes

\begin{align*}
C & ::= \mathbf{0} \mid C \parallel C \mid \nu(n:T).C \mid n[(O)] \mid n[n,F] \mid n\langle \rangle \\
O & ::= M, F \\
M & ::= l = m, \ldots, l = m \\
F & ::= l = f, \ldots, l = f \\
m & ::= \zeta(n:T).\lambda(x:T, \ldots, x:T).t \\
f & ::= \zeta(n:T).\lambda().v \\
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 v.l(v, \ldots, v) \mid n.l := v \mid \text{currentthread} \\
 & \mid \text{new } n \mid \text{new} \langle \rangle \\
v & ::= x \mid n
\end{align*}
Semantics

- given in various “stages”
  - internal (component-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
Internal steps

- $o_1$ creates an internal object $o_3$ (assume: thread $n$ visits $o_1$)

$$c[(F,M)] \parallel n\langle \text{let } x : c = \text{new } c \text{ in } t \rangle \sim\rightarrow$$

$$c[(F,M)] \parallel \nu o_3 : c. (n\langle \text{let } x : c = o_3 \text{ in } t \rangle \parallel o_3[c,F]) \ldots$$
Semantics: Internal steps

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

\[
\begin{align*}
  n\langle \text{let } x : T = v \text{ in } t \rangle & \rightsquigarrow n\langle t[v/x] \rangle \\
  c[F, M] \parallel n\langle \text{let } x : c = \text{new } c \text{ in } t \rangle & \rightsquigarrow c[F, M] \parallel \nu o : c. (n\langle \text{let } x : c = o \text{ in } t \rangle \parallel o[c, F]) \\
  c[F, M] \parallel o[c, F'] \parallel n\langle \text{let } x : T = o.l(\vec{v}) \text{ in } t \rangle & \rightarrow c[F, M] \parallel o[c, F'] \parallel n\langle \text{let } x : T = M.l(o)(\vec{v}) \text{ in } t \rangle
\end{align*}
\]
Semantics: External steps

- "typed" operational semantics
- i.e., labeled steps between typing judgments: $\Delta \vdash P : \Theta$
  - $\Delta =$ "assumptions"
    - names assumed present in the rest
  - $\Theta =$ "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

\[ o \in \Delta \]

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

\[ ; \Delta, \Theta \vdash v : T \]

\[ \Delta \vdash C \parallel n\langle \text{let } x : T = \text{block in } t \rangle : \Theta \xrightarrow{n\langle \text{return}(v) \rangle ?} \Delta \vdash C \parallel n\langle t[v/x] \rangle : \Theta \]
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: “v”
External steps: Scoping

internal \( o_3 \) is sent outside, e.g., as argument of method call to external \( o_2 \)
External steps: Scoping

\[ \Delta \vdash \nu o_3 : c_3 \cdot (n\langle o_2 . l( o_3 ) ; t \rangle \parallel o_1 [ \ldots ] ) : \Theta \quad \text{\(\nu o_3 : c_3 \cdot n\langle \text{call } o_2 . l( o_3 ) \rangle!\)} \]
\[ \Delta \vdash n\langle \text{block}; t \rangle \parallel o_3 [ \ldots ] : \Theta, \quad o_3 : c_3 \]
Introduction

Calculus

Classes and observable behavior

Object connectivity

Conclusion
F-A in an object-based conc. setting

- [3]: for the concurrent $\nu$-calculus
- notion of observation: may-testing equivalence.\footnote{actually: they use may-preorder.}
  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
What changes?

- **classes** are 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!
Introduction

Calculus

Classes and observable behavior

Object connectivity

Conclusion
Completeness: line of argument

- goal: if $C_1 \equiv_{\text{obs}} C_2$, then $C_1 \equiv_{\mathcal{D}} C_2$
- so, given a legal trace $s \in \llbracket C_1 \rrbracket_{\mathcal{D}}$, do
  - construct a complementary context $C_{\bar{s}}$
  - composition: program + context do the observation
    $$C_{\bar{s}}[C_1] \longrightarrow^* \text{success}$$
- observational equivalence: $C_2$ can do that, too:
  $$C_{\bar{s}}[C_2] \longrightarrow^* \text{success}$$
- decomposition:\[2\] $s \in \llbracket C_2 \rrbracket_{\mathcal{D}}$

$\Rightarrow$ problems for completeness (apart from technicalities)

1. definability $\Rightarrow$ what are legal traces?
2. what can be observed/distinguished?

\[2\]That $s$ is a trace of $C_2$ by decomposition is not a direct consequence. I ignore that here.
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?

\[ \nu o_1.\text{creates } o_1!. \; \nu o_3.\text{creates } o_3!. \; n'(\text{call } o_2.\text{f}(o_1, o_3))? \]

impossible!
Acquaintance

- $o_1$ 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 order of interaction
- connectivity or “acquaintance” is dynamic
- the only one to make $o_1$ and $o_3$ acquainted: the component
Dynamic acquaintance

\[ O_1 \]

\[ O_2 \]

\[ O_3 \]
Dynamic acquaintance

\[ o_3! \]

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

- no scope extrusion from the (global) 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
  - keep track of (the worst-case) of connectivity
  - 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

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

- Assumption context: \( E_\Delta \subseteq \Delta \times (\Delta + \Theta) = \text{pairs of objects} \)
- 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 \( o_2 \in \Theta: E_\Delta \vdash o_1 \Leftrightarrow ; \leftarrow o_2 \))
Scoping & lazy instantiation

- object names get known “on the other side” ⇒ scope extrusion
- external classes ⇒ cross border instantiation
  - instance created “on the other side”
  - reference kept “at this side”
- instantiation itself: not observable ⇒ lazy instantiation

\[ \Delta \vdash \nu(o_2 : c_2)(n\langle o_1 . l(o_2) ; t \rangle \parallel o_2[\ldots] \parallel \ldots) : \Theta, c_2 : T_2 \]

- label \( a = \nu(\Theta', \Delta'). n\langle \text{call } o_2 . l(\vec{v}) \rangle! \)
Scoping & lazy instantiation

- object names get known “on the other side” ⇒ scope extrusion
- external classes ⇒ cross border instantiation
  - instance created “on the other side”
  - reference kept “at this side”
- instantiation itself: not observable ⇒ lazy instantiation

\[ \Delta \vdash \nu(o_2:c_2)(n\langle o_1.l(o_2); t \rangle \parallel o_2[\ldots] \parallel \ldots) : \Theta, c_2:T_2 \]

\[ \nu(o_2:c_2).n\langle \text{call } o_1.l(o_2) \rangle! \]

- label \( a = \nu(\Theta', \Delta'). n\langle \text{call } o_2.l(\bar{v}) \rangle! \)
Scoping & lazy instantiation

- object names get known “on the other side” \(\Rightarrow\) scope extrusion
- external classes \(\Rightarrow\) cross border instantiation
  - instance created “on the other side”
  - reference kept “at this side”
- instantiation itself: not observable \(\Rightarrow\) lazy instantiation

\[
\Delta \vdash \nu(o_2:c_2)(n\langle o_1.l(o_2); t \rangle \parallel o_2[\ldots] \parallel \ldots) : \Theta, c_2:T_2
\]

\[
\nu(o_2:c_2).n\langle call o_1.l(o_2)\rangle!
\]

\[
\Delta \vdash n\langle block; t \rangle \parallel o_2[\ldots] \parallel \ldots : \Theta, c_2:T_2, o_2:c_2
\]

- label \(a = \nu(\Theta', \Delta'). n\langle call o_2.l(\vec{v})\rangle!\)
Scoping & lazy instantiation

- object names get known “on the other side” \( \Rightarrow \) scope extrusion
- external classes \( \Rightarrow \) cross border instantiation
  - instance created “on the other side”
  - reference kept “at this side”
- instantiation itself: not observable \( \Rightarrow \) lazy instantiation

\[
\Delta, c_2 : T_2 \vdash \nu(o_2 : c_2)(n(o_1 . l(o_2) ; t) \parallel \ldots) : \Theta
\]

- label \( \lambda a = \nu(\Theta', \Delta'). n\langle \text{call } o_2 . l(\vec{v}) \rangle! \)
Scoping & lazy instantiation

- object names get known “on the other side” ⇒ scope extrusion
- external classes ⇒ cross border instantiation
  - instance created “on the other side”
  - reference kept “at this side”
- instantiation itself: not observable ⇒ lazy instantiation

\[
\Delta, c_2 : T_2 \vdash \nu(o_2 : c_2)(n\langle o_1.l(o_2); t \rangle \ || \ \ldots) : \Theta
\]

\[
\nu(o_2 : c_2).n\langle \text{call } o_1.l(o_2) \rangle
\]

- label \( a = \nu(\Theta', \Delta'). n\langle \text{call } o_2.l(\vec{v}) \rangle ! \)
Scoping & lazy instantiation

- object names get known “on the other side” ⇒ scope extrusion
- external classes ⇒ cross border instantiation
  - instance created “on the other side”
  - reference kept “at this side”
- instantiation itself: not observable ⇒ lazy instantiation

\[
\Delta, c_2 : T_2 \vdash \nu(o_2 : c_2)(n(o_1 . l(o_2) ; t) \parallel \ldots) : \Theta
\]

\[
\nu(o_2 : c_2) . n(\text{call } o_1 . l(o_2)) !
\]

\[
\Delta, c_2 : T_2, o_2 : T_2 \vdash n(\text{block} ; t) \parallel \ldots : \Theta
\]

- label \( a = \nu(\Theta' , \Delta') . n(\text{call } o_2 . l(\vec{v})) ! \)
Legal traces

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

\[
\Delta; E_\Delta \vdash r \triangleright s : \text{trace } \Theta; E_\Theta
\]
Legal traces: incoming call

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

\[
\text{lots of conditions} \quad \Delta; \; \hat{E}_\Delta \vdash \quad r \; a \triangleright s \quad : \; \text{trace } \Theta; \; \hat{E}_\Theta
\]

\[
\Delta; \; E_\Delta \vdash \quad r \; \triangleright \; a \; s \quad : \; \text{trace } \Theta; \; E_\Theta
\]
"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")
Incoming call: acquaintance

- let \( a = n\langle \text{call } o_2. l(\vec{v}) \rangle \)?

\[
\hat{E}_\Theta = E_\Theta + o_2 \leftrightarrow \vec{v}
\]

\[
\Delta; E_\Delta \vdash r \ a \triangleright s : \text{trace}\Theta; \hat{E}_\Theta \quad E_\Delta \vdash v_i \Leftrightarrow v_j
\]

\[
\Delta; E_\Delta \vdash r \triangleright a \ s : \text{trace}\Theta
\]

- caller anonymous \( \Rightarrow \) not mentioned in label
- nonetheless: needed for bookkeeping: to return to the same caller

\( \Rightarrow \) remembered in the history
Incoming call: Who’s the caller?

- let \( a = n\langle \text{call} \ o_2. \ell(\vec{v}) \rangle \)?

\[
\dot{E}_\Theta = E_\Theta + o_2 \leftrightarrow \vec{v}
\]

\[
\Delta; \ E_\Delta \vdash r \ a \quad \triangleright \ s : \text{trace} \Theta; \ \dot{E}_\Theta \quad \quad E_\Delta \vdash v_i \equiv v_j
\]

\[
\Delta; \ E_\Delta \vdash r \triangleright a \ s : \text{trace} \Theta
\]

- caller anonymous \( \Rightarrow \) not mentioned in label
- nonetheless: needed for bookkeeping: to return to the same caller

\( \Rightarrow \) remembered in the history
Incoming call: Who’s the caller?

- let \( a = n \langle \text{call } o_2.1(\bar{v}) \rangle \)?

\[
\dot{E}_\Theta = E_\Theta + o_2 \leftrightarrow \bar{v}
\]

\[
\Delta; E_\Delta \vdash r\ a_{o_1} \triangleright s : \text{trace}\Theta; \dot{E}_\Theta
\]

\[
\Delta; E_\Delta \vdash o_1 \iff \bar{v}, o_2 : \Theta; E_\Theta
\]

\[
\Delta; E_\Delta \vdash r \triangleright a\ s : \text{trace}\Theta
\]

- caller **anonymous** \( \Rightarrow \) not mentioned in label
- nonetheless: needed for **bookkeeping**: to return to the same caller

\( \Rightarrow \) **remembered** in the history
Incoming call: scoping

- object names get known “on the other side” \( \Rightarrow \) **scope extrusion**
- external classes \( \Rightarrow \) **cross border** instantiation
  - instance created “on the other side”
  - reference kept “at this side”
- instantiation itself: not observable \( \Rightarrow \) **lazy instantiation**
- label \( a = \nu(\Delta', \Theta'). \ n\langle call \ o_2.l(\vec{v})\rangle? \)

\[
\begin{align*}
\Delta & \vdash o_1 : c_1 \\
\Theta; \mathcal{E}_\Theta = \emptyset; E_\Theta + (\Theta'; o_2 \leftrightarrow \vec{v}) & \quad \Delta; \mathcal{E}_\Delta = \Delta; E_\Delta + \Delta'; o_1 \leftrightarrow (\Delta', \Theta') \\
\text{dom}(\Delta', \Theta') & \subseteq \text{fn}(n\langle call \ o_2.l(\vec{v})\rangle) \\
\Delta; \mathcal{E}_\Delta & \vdash o_1 \leftrightarrow \vec{v}, o_2 : \Theta \\
\Delta; \mathcal{E}_\Delta & \vdash r \ a_{o_1} \triangleright s : \text{trace} \Theta; \mathcal{E}_\Theta \\
\Delta; E_\Delta & \vdash r \triangleright a \ s : \text{trace} \Theta; E_\Theta
\end{align*}
\]
Legal traces: balance

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

\[ \text{pop } n \text{ r} = \nu(\Theta'). n\langle \text{call } o_2.l(\vec{v})! \rangle \]
\[ \Delta \vdash r \triangleright \nu(\Delta'). n\langle \text{return}(\nu)\rangle? : \Theta \]

- before a return: there must have been an outgoing call
- \text{pop} picks out the last “matching” call
Incoming comm.: the full story

\[ a = \nu(\Delta', \Theta'). \; n\langle \text{call } o_2.\text{l}(\vec{v})\rangle? \quad \Delta \vdash o_1 : c_1 \quad \Delta \vdash r \triangleright a : \Theta \]

\[ \delta; \; \hat{E}_\Theta = \Theta; \; E_\Theta + (\Theta'; o_2 \leftrightarrow \vec{v}, n \leftrightarrow o_2) \quad \hat{\Delta}; \; \hat{E}_\Delta = \Delta; \; E_\Delta + \Delta'; o_1 \leftrightarrow (\Delta', \Theta') \setminus n \]

\[ \delta \vdash o_2 :: [\ldots, l: \ll T \rightarrow T, \ldots] \quad \hat{\Delta} \vdash n: \text{thread} \quad \delta \vdash n: \text{thread} \quad \hat{\Delta}, \delta \vdash \vec{v} : \tilde{T} \]

\[ \text{dom}(\Delta', \Theta') \subseteq \text{fn}(n\langle \text{call } o_2.\text{l}(\vec{v})\rangle) \]

\[ \hat{\Delta}; \; \hat{E}_\Delta \vdash n \Leftarrow o_1 \Leftarrow \vec{v}, o_2 : \Theta \quad \hat{\Delta}; \; \hat{E}_\Delta \vdash r \; a_{o_1} \triangleright s : \text{trace} \delta; \; \hat{E}_\Theta \]

\[ \Delta; \; E_\Delta \vdash r \triangleright a \; s : \text{trace} \delta; \; E_\Theta \]
Definability

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

  induction on the derivation for
  $\Delta; E_\Delta \vdash r \triangleright s : trace \Theta; E_\Theta$

$\Rightarrow$ construct the program backwards!

actions on the commitment context $E_\Theta$:

  - $E_\Theta$: 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/synchronization

- for example outgoing call \( a = \nu(\Theta', \Delta'). n\langle \text{call } o_2.l(\bar{v}) \rangle! \)
- we know: afterwards

\[
\hat{C}_s = n\langle o_1 \text{ blocks for } o_2; t' \rangle \parallel C'_s
\]
\[
\hat{E}_\Theta = E_\Theta + o_1 \leftrightarrow \Theta'
\]

- construct component \( \hat{C}_s \) before the call:

\[
\hat{C}_s = C'_s \parallel n\langle t^o_{\text{sync}}(\Theta', \bar{a}); o_2.l(\bar{v}) \rangle
\]

where

\[
t^o_{\text{sync}}(\Theta', \bar{a}) \triangleq ( \mid \text{let } \bar{x}:\bar{c} = \text{new}(\Theta') \\
\mid \text{in } \text{pick}(\bar{x}); \\
\mid \text{self}.\text{propagate}(\bar{x}); \\
\mid \bar{a} \triangleright | )
\]
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 (in first approximation)
  - 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
Overview

Introduction

Calculus

Classes and observable behavior

Object connectivity

Conclusion
Conclusions

- are classes good composition units?
- what about cloning?
- lock-synchronization
- subtype polymorphism & subclassing
References I

_A Theory of Objects_.  

A concurrent object calculus: Reduction and typing.  

A fully abstract may testing semantics for concurrent objects.  

Fully abstract models of typed _λ_-calculi.  

A calculus of mobile processes, part I/II.  

Observable properties of higher-order functions that dynamically create local names, or: What's new.  

LCF considered as a programming language.  
References II