# Game Semantics for Interface Middleweight Java

Nikos Tzevelekos
Queen Mary University of London

joint work with Andrzej Murawski (University of Warwick)

Semantics and Verification of OO languages, Shonan, Sep 2015

*supported by a Royal Academy of Engineering research fellowship*

# What this talk is about

- a denotational model for a core fragment of Java
- the model is complete wrt contextual equivalence
- first of this kind and constructed in game semantics

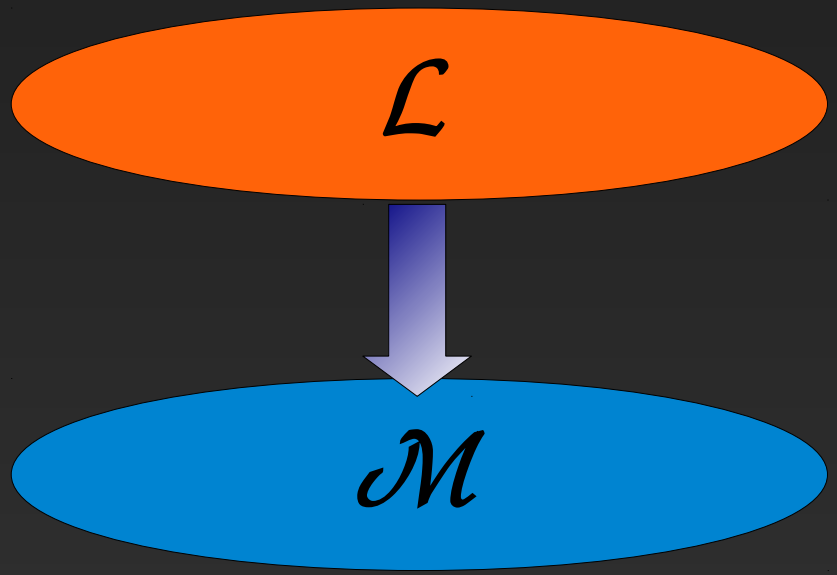
# Denotational Semantics

Translate programs into a formal model (of 'functions'):

- abstract mathematical description
- free-of-syntax meaning

program: $P$

denotation: $[\![P]\!]$



# Power of denotations

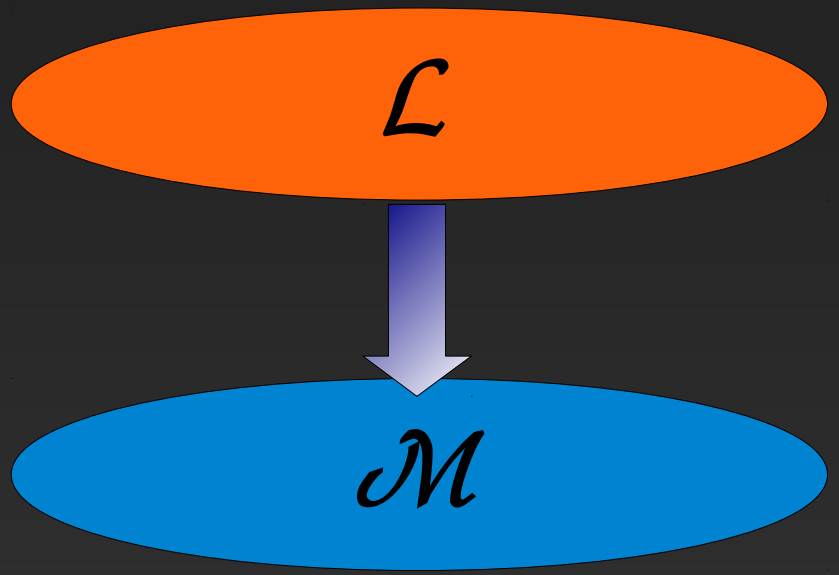
Abstract away from implementation details

Compositional translation:

- modelling of components in isolation
- modularity

program: $P$

denotation: $[\![P]\!]$




Useful for understanding & analysing programs

# Full abstraction criterion

*Ideally, a mathematical model exactly describes the meaning of elements in a system (here: programs)*

Full abstraction (equational):

$$P \cong P' \iff [\![P]\!] = [\![P']\!]$$

read $P \cong P'$ as:

*same observable behaviour in every context*

i.e. for all closing contexts $C$,

$C[P]$ terminates $\iff$ $C[P']$ terminates

# Full abstraction criterion

*Ideally, a mathematical model exactly describes the meaning of elements in a system (here: programs)*

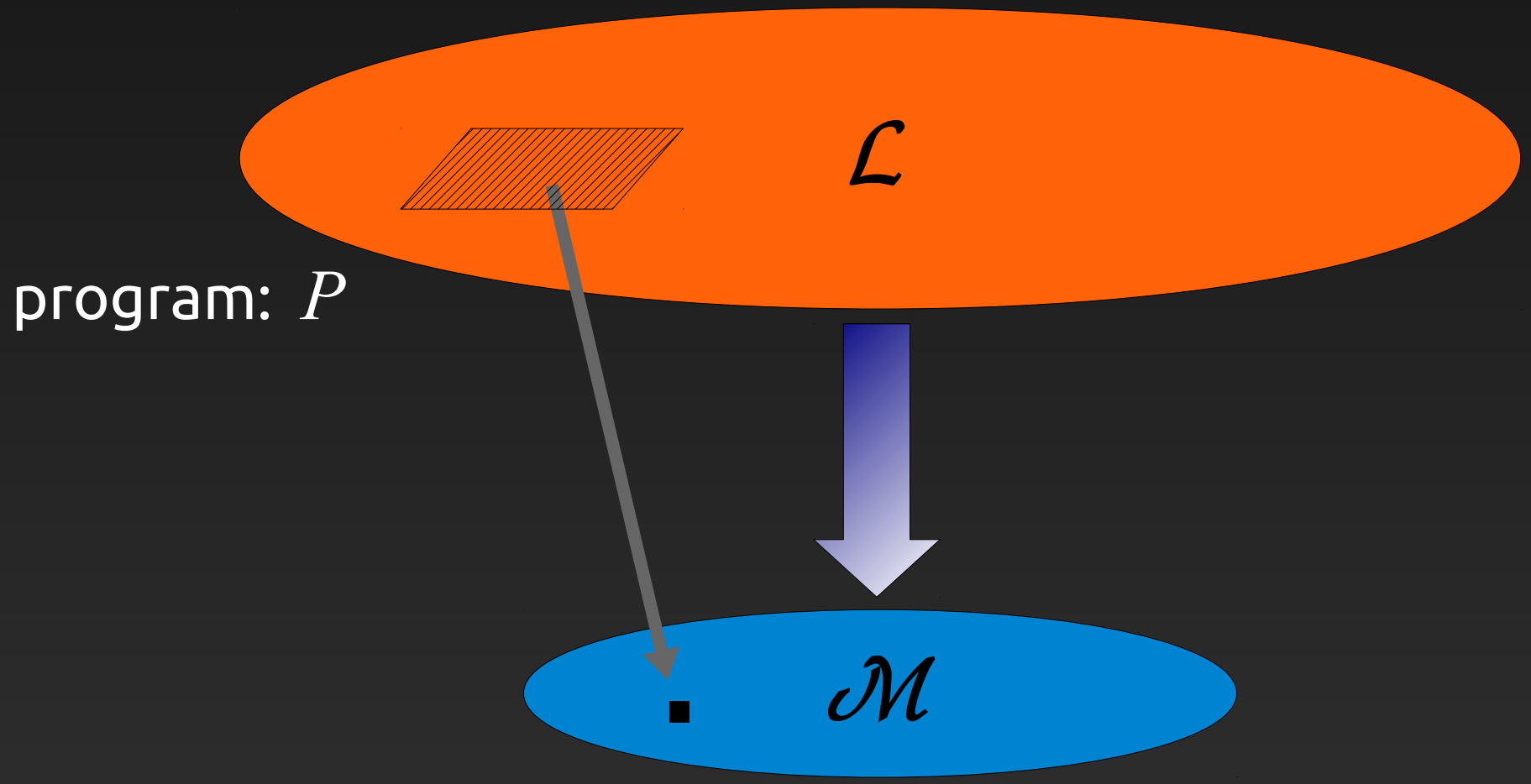
Full abstraction (equational):

$$P \cong P' \iff [\![P]\!] = [\![P']\!]$$

Full abstraction (inequational):

$$P \sqsubseteq P' \iff [\![P]\!] \subseteq [\![P']\!]$$

# Full abstraction pictorially



# The quest for full abstraction

1977 [Milner, Plotkin]:

- Formulation of the problem
- Functions cannot capture sequentiality (PCF)

*TCS'77*

**FULLY ABSTRACT MODELS OF TYPED λ-CALCULI**

Robin MILNER
*Computer Science Department, Edinburgh University, Edinburgh, Scotland*

Communicated by Maurice Nivat
Received October 1975
Revised June 1976

**Abstract.** A semantic interpretation $\mathscr{A}$ for a programming language $L$ is fully abstract if, whenever $\mathscr{A}[\![\mathscr{C}[M]]\!] \sqsubseteq \mathscr{A}[\![\mathscr{C}[N]]\!]$ for two program phrases $M, N$ and for all program contexts $\mathscr{C}[\ ]$, it follows that $\mathscr{A}[\![M]\!] \sqsubseteq \mathscr{A}[\![N]\!]$. A model $\mathscr{M}$ for the language is fully abstract if the natural interpretation $\mathscr{A}$ of $L$ in $\mathscr{M}$ is fully abstract.

We show that under certain conditions there exists, for an extended typed λ-calculus, a unique fully abstract model.

## 1. Introduction

We are concerned with the problem of finding, for a programming language, a denotational semantic definition which is not over-generous in a certain sense. We can describe quite informally what we mean by 'over-generosity'. Suppose that $L$ is the set of well-formed phrases of the language. Often it is the case that not every such phrase is a whole program; for example, a procedure declaration may not be one, though of course may be part of one.

*TCS'77*

**LCF CONSIDERED AS A PROGRAMMING LANGUAGE**

G.D. PLOTKIN
*Department of Artificial Intelligence, University of Edinburgh, Hope Park Square, Meadow Lane, Edinburgh EH8 9NW, Scotland*

Communicated by Robin Milner
Received July 1975

**Abstract.** The paper studies connections between denotational and operational semantics for a simple programming language based on LCF. It begins with the connection between the behaviour of a program and its denotation. It turns out that a program denotes ⊥ in any of several possible semantics iff it does not terminate. From this it follows that if two terms have the same denotation in one of these semantics, they have the same behaviour in all contexts. The converse fails for all the semantics. If, however, the language is extended to allow certain parallel facilities, behavioural equivalence does coincide with denotational equivalence in one of the semantics considered, which may therefore be called "fully abstract". Next a connection is given which actually determines the semantics up to isomorphism from the behaviour alone. Conversely, by allowing further parallel facilities, every r.e. element of the fully abstract semantics becomes definable, thus characterising the programming language, up to interdefinability, from the set of r.e. elements of the domains of the semantics.

## 1. Introduction

We present here a study of some connections between the operational and

# The quest for full abstraction

1977 [Milner, Plotkin]:

- Formulation of the problem
- Functions cannot capture sequentiality (PCF)

1980-90's:

- Function stability [Berry, Bucciarelli, Erhard]
- Sequential algorithms [Berry, Currien]

1993 [AJM, HO/N *]: Game semantics (PCF)

- 'Functions' with operational content (*games*)

* Abramsky, Jagadeesan, Malacaria; Hyland, Ong; Nickau

# From PCF to realistic languages

*Full Abstraction for PCF* (early 90's)

First stage (1993-2004)

- Models for various variants of Idealized Algol (non-determinism, exceptions, call-by-value, …)

Nominal game semantics (2004-)

- Fragments of ML, now Java (IMJ)
- Fundamental difference: use of resources (*names*)

# Interface Middleweight Java (IMJ)

Object calculus based on MJ [Bierman, Parkinson, Pitts]

- Objects, inheritance, casting, interfaces

# Interface Middleweight Java (IMJ)

*Types* $\quad \theta ::= \text{void} \mid \text{int} \mid \mathcal{I}$

*Interface definitions*

$$\Theta ::= \phi \mid (\mathsf{f} : \theta), \Theta \mid (\mathsf{m} : \overline{\theta} \rightarrow \theta), \Theta$$

*Interface tables*

$$\Delta ::= \phi \mid (\mathcal{I} : \Theta), \Delta \mid (\mathcal{I}\langle\mathcal{I}\rangle : \Theta), \Delta$$

Object calculus based on MJ [Bierman, Parkinson, Pitts]

- Objects, inheritance, casting, interfaces

# Interface Middleweight Java (IMJ)

interface ident.

*Types* $\theta ::= \text{void} \mid \text{int} \mid \mathcal{I}$

*Interface definitions*

$$\Theta ::= \phi \mid (\mathsf{f} : \theta), \Theta \mid (\mathsf{m} : \overline{\theta} \rightarrow \theta), \Theta$$

*Interface tables*

$$\Delta ::= \phi \mid (\mathcal{I} : \Theta), \Delta \mid (\mathcal{I}\langle\mathcal{I}\rangle : \Theta), \Delta$$

Object calculus based on MJ [Bierman, Parkinson, Pitts]

- Objects, inheritance, casting, interfaces

# Interface Middleweight Java (IMJ)

*Types* $\theta ::= \text{void} \mid \text{int} \mid \mathcal{I}$

*Interface definitions*

$\Theta ::= \phi \mid (\mathsf{f} : \theta), \Theta \mid (\mathsf{m} : \overline{\theta} \to \theta), \Theta$

*Interface tables*

$\Delta ::= \phi \mid (\mathcal{I} : \Theta), \Delta \mid (\mathcal{I}\langle\mathcal{I}\rangle : \Theta), \Delta$

interface ident.

field identifiers

method identif.

Object calculus based on MJ [Bierman, Parkinson, Pitts]

- Objects, inheritance, casting, interfaces

# Interface Middleweight Java (IMJ)

*Terms*

$$M ::= \text{skip} \mid a \mid \text{null} \mid x \mid i \mid M \oplus M \mid \text{if } M\, M\, M$$
$$\mid \text{let } x = M \text{ in } M \mid M = M \mid (\mathcal{I})M$$
$$\mid \text{new}(x : \mathcal{I};\ \mathcal{M}) \mid M.\text{f} \mid M.\text{f} := M \mid M.\text{m}(\overline{M})$$

*Method implementations* $\mathcal{M} ::= \phi \mid (\text{m} : \lambda \overline{x}.M),\ \mathcal{M}$

# Interface Middleweight Java (IMJ)

$\mathcal{M} = \{ \mathsf{m}_i : \lambda \bar{x}_i . M_i \mid 1 \le i \le n \}$

$$\frac{\Delta|\Gamma \vdash M : \mathsf{int} \quad \Delta|\Gamma \vdash M', M'' : \theta}{\Delta|\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ M'\ \mathsf{else}\ M'' : \theta}$$

$$\frac{\bigwedge_{i=1}^{n} (\Delta|\Gamma \uplus \{\vec{x}_i : \vec{\theta}_i\} \vdash M_i : \theta_i)}{\Delta|\Gamma \vdash \mathcal{M} : \{\mathsf{m}_i : \vec{\theta}_i \to \theta_i \mid 1 \le i \le n\}}$$

$$\frac{\Delta|\Gamma \vdash M, M' : I}{\Delta|\Gamma \vdash M = M' : \mathsf{int}} \qquad \frac{\Delta|\Gamma \vdash M : \mathsf{void} \quad \Delta|\Gamma \vdash M' : \theta}{\Delta|\Gamma \vdash M; M' : \theta}$$

$$\frac{\Delta|\Gamma \vdash M : I \quad \Delta|\Gamma \vdash M' : \theta}{\Delta|\Gamma \vdash M.\mathsf{f} := M' : \mathsf{void}}\ \Delta(I).\mathsf{f}=\theta$$

$$\frac{\Delta|\Gamma \vdash M : I}{\Delta|\Gamma \vdash M.\mathsf{f} : \theta}\ \Delta(I).\mathsf{f}=\theta \qquad \frac{\Delta|\Gamma \vdash M : I'}{\Delta|\Gamma \vdash (I)M : I}\ \begin{array}{l}\Delta \vdash I \le I' \\ \vee I' \le I\end{array}$$

$$\frac{\Delta|\Gamma, x : I \vdash \mathcal{M} : \Theta}{\Delta|\Gamma \vdash \mathsf{new}\{\, x : I; \mathcal{M}\,\} : I}\ \Delta(I) \upharpoonright Meths = \Theta$$

$$\frac{\Delta|\Gamma \vdash M : I \quad \bigwedge_{i=1}^{n} (\Delta|\Gamma \vdash M_i : \theta_i)}{\Delta|\Gamma \vdash M.\mathsf{m}(M_1, \cdots, M_n) : \theta}\ \Delta(I).\mathsf{m}=\vec{\theta}\to\theta \qquad \frac{\Delta|\Gamma \vdash M : \theta' \quad \Delta|\Gamma, x : \theta' \vdash M' : \theta}{\Delta|\Gamma \vdash \mathsf{let}\ x = M\ \mathsf{in}\ M' : \theta}$$

*Terms*

$$M ::= \mathsf{skip} \mid a \mid \mathsf{null} \mid x \mid i \mid M \oplus M \mid \mathsf{if}\ M\ M\ M$$
$$\mid \mathsf{let}\ x = M\ \mathsf{in}\ M \mid M = M \mid (\mathcal{I})M$$
$$\mid \mathsf{new}(x : \mathcal{I}; \mathcal{M}) \mid M.\mathsf{f} \mid M.\mathsf{f} := M \mid M.\mathsf{m}(\bar{M})$$

*Method implementations* $\mathcal{M} ::= \phi \mid (\mathsf{m} : \lambda \bar{x}.M), \mathcal{M}$

# IMJ: operational semantics

$$S,\ M \ \longrightarrow\ S',\ M'$$

$S$ stores object names + their types and values

Obj : set of obj. names

$$S,\ \text{let}\ x = v\ \text{in}\ M \ \longrightarrow\ S,\ M[v/x]$$

$$S,\ a = a' \ \longrightarrow\ S,\ 0/1 \qquad a,a' \in \text{Obj}$$

$$S,\ (\mathcal{I})a \ \longrightarrow\ S,\ a \qquad \text{if}\ \ S(a) : \mathcal{I}'\ \text{and}\ \ \mathcal{I}' \leq \mathcal{I}$$

$$S,\ \text{new}(x{:}\mathcal{I};\ \mathcal{M}) \ \longrightarrow\ S \uplus \{(a,\ \mathcal{I},\ (V_{\mathcal{I}},\ \mathcal{M}[a/x])\},\ a$$

$V_{\mathcal{I}}$: default field values

# IMJ example*

$M_1$ : let $u$ = new(Var$_{Emp}$) in
new(Cell; $\mathcal{M}_1$) : Cell

$\mathcal{M}_1$ : get: $\lambda().\ u.$val,
set: $\lambda y.\ u.$val := $y$

$\Delta$ = Empty: $\emptyset$,
Cell: (get: void $\rightarrow$ Empty,
set: Empty $\rightarrow$ void),
Var$_{Emp}$: (val: Empty),
Var$_{Int}$: (val: int)

* Koutavas & Wand, 2007

# IMJ example*

$\Delta$ = Empty: $\emptyset$,
Cell: (get: void $\rightarrow$ Empty,
set: Empty $\rightarrow$ void),
$\text{Var}_{Emp}$: (val: Empty),
$\text{Var}_{Int}$: (val: int)

$M_1$ : let $u = \text{new}(\text{Var}_{Emp})$ in
new(Cell; $\mathcal{M}_1$) : Cell

$\mathcal{M}_1$ : get: $\lambda().\ u.\text{val}$,
set: $\lambda y.\ u.\text{val} := y$

$M_2$ : let $b = \text{new}(\text{Var}_{Int})$ in
let $u_1 = \text{new}(\text{Var}_{Emp})$ in
let $u_2 = \text{new}(\text{Var}_{Emp})$ in
new(Cell; $\mathcal{M}_2$) : Cell

$\mathcal{M}_2$ : get: $\lambda()$. if $b.\text{val}$
then $b.\text{val} := 0\,;\ u_1.\text{val}$
else $b.\text{val} := 1\,;\ u_2.\text{val}$,
set: $\lambda y.\ u_1.\text{val} := y\,;$
$u_2.\text{val} := y$

* Koutavas & Wand, 2007

# IMJ example*

$M_1$ : let $u$ = new($\text{Var}_{Emp}$) in
new(Cell; $\mathcal{M}_1$) : Cell

$\mathcal{M}_1$ : get: $\lambda().\ u.\text{val}$,
set: $\lambda y.\ u.\text{val} := y$

$\Delta$ = Empty: $\emptyset$,
Cell: (get: void $\rightarrow$ Empty,
set: Empty $\rightarrow$ void),
$\text{Var}_{Emp}$: (val: Empty),
$\text{Var}_{Int}$: (val: int)

$$M_1 \cong M_2$$

$M_2$ : let $b$ = new($\text{Var}_{Int}$) in
let $u_1$ = new($\text{Var}_{Emp}$) in
let $u_2$ = new($\text{Var}_{Emp}$) in
new(Cell; $\mathcal{M}_2$) : Cell

$\mathcal{M}_2$ : get: $\lambda()$. if $b$.val
then $b.\text{val} := 0$; $u_1.\text{val}$
else $b.\text{val} := 1$; $u_2.\text{val}$,
set: $\lambda y.\ u_1.\text{val} := y$;
$u_2.\text{val} := y$

* Koutavas & Wand, 2007

# IMJ example: game semantics

$M_1$ : let $u$ = new($\mathrm{Var}_{Emp}$) in
new(Cell; $\mathcal{M}_1$) : Cell

$\mathcal{M}_1$ : get: $\lambda().\ u.\mathrm{val}$,
set: $\lambda y.\ u.\mathrm{val} := y$

$\Delta$ = Empty: $\emptyset$,
Cell: (get: void $\to$ Empty,
set: Empty $\to$ void),
$\mathrm{Var}_{Emp}$: (val: Empty),
$\mathrm{Var}_{Int}$: (val: int)

**O** **P** **O** **P**

$[\![M_1]\!] = * \ n^{\Sigma_0}$ ( call $n$.get()$^{\Sigma_0}$ ret $n$.get(nul)$^{\Sigma_0}$ )*
call $n$.set($n_1$)$^{\Sigma_1}$ ret $n$.set()$^{\Sigma_1}$
( call $n$.get()$^{\Sigma_1}$ ret $n$.get($n_1$)$^{\Sigma_1}$ )*
call $n$.set($n_2$)$^{\Sigma_2}$ ret $n$.set()$^{\Sigma_2}$ ...

$\Sigma_i = \{\ n \mapsto (\mathrm{Cell}, \emptyset)\ \} \cup \{\ n_j \mapsto (\mathrm{Empty}, \emptyset),\ 1 \le j \le i\ \}$

# IMJ example: game semantics

$M_1$ : let $u$ = new($\mathrm{Var}_{Emp}$) in
new(Cell; $\mathcal{M}_1$) : Cell

$\mathcal{M}_1$ : get: $\lambda().\ u.\mathrm{val}$,
set: $\lambda y.\ u.\mathrm{val} := y$

$\Delta$ = Empty: $\emptyset$,
Cell: (get: void $\to$ Empty,
set: Empty $\to$ void),
$\mathrm{Var}_{Emp}$: (val: Empty),
$\mathrm{Var}_{Int}$: (val: int)

**O** **P** **O** **P**

$[\![M_1]\!] = * \ n^{\Sigma_0}$ ( call $n$.get()$^{\Sigma_0}$ ret $n$.get(nul)$^{\Sigma_0}$ )*
call $n$.set($n_1$)$^{\Sigma_1}$ ret $n$.set()$^{\Sigma_1}$
( call $n$.get()$^{\Sigma_1}$ ret $n$.get($n_1$)$^{\Sigma_1}$ )*
call $n$.set($n_2$)$^{\Sigma_2}$ ret $n$.set()$^{\Sigma_2}$ ... $= [\![M_2]\!]$

$\Sigma_i = \{\ n \mapsto (\mathrm{Cell}, \emptyset)\ \} \cup \{\ n_j \mapsto (\mathrm{Empty}, \emptyset),\ 1 \le j \le i\ \}$

# Game Semantics

Computation is modelled as a 2-player game between:

- *Opponent* (the environment, $O$ )
- *Proponent* (the program, $P$ )

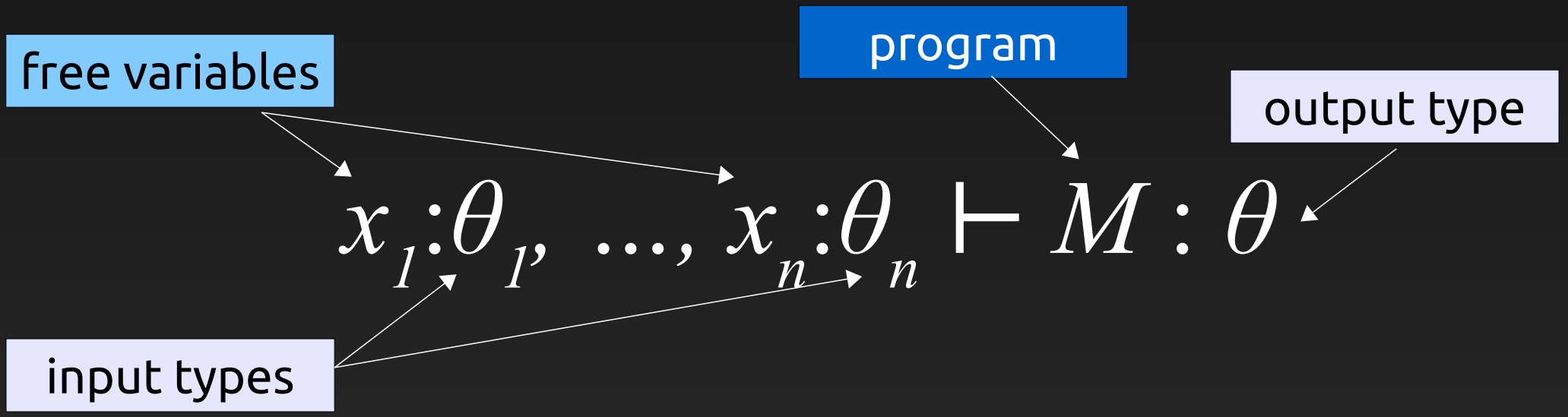
Qualitative games ( ≠ Game Theory)

Computations = *plays* of a specified game

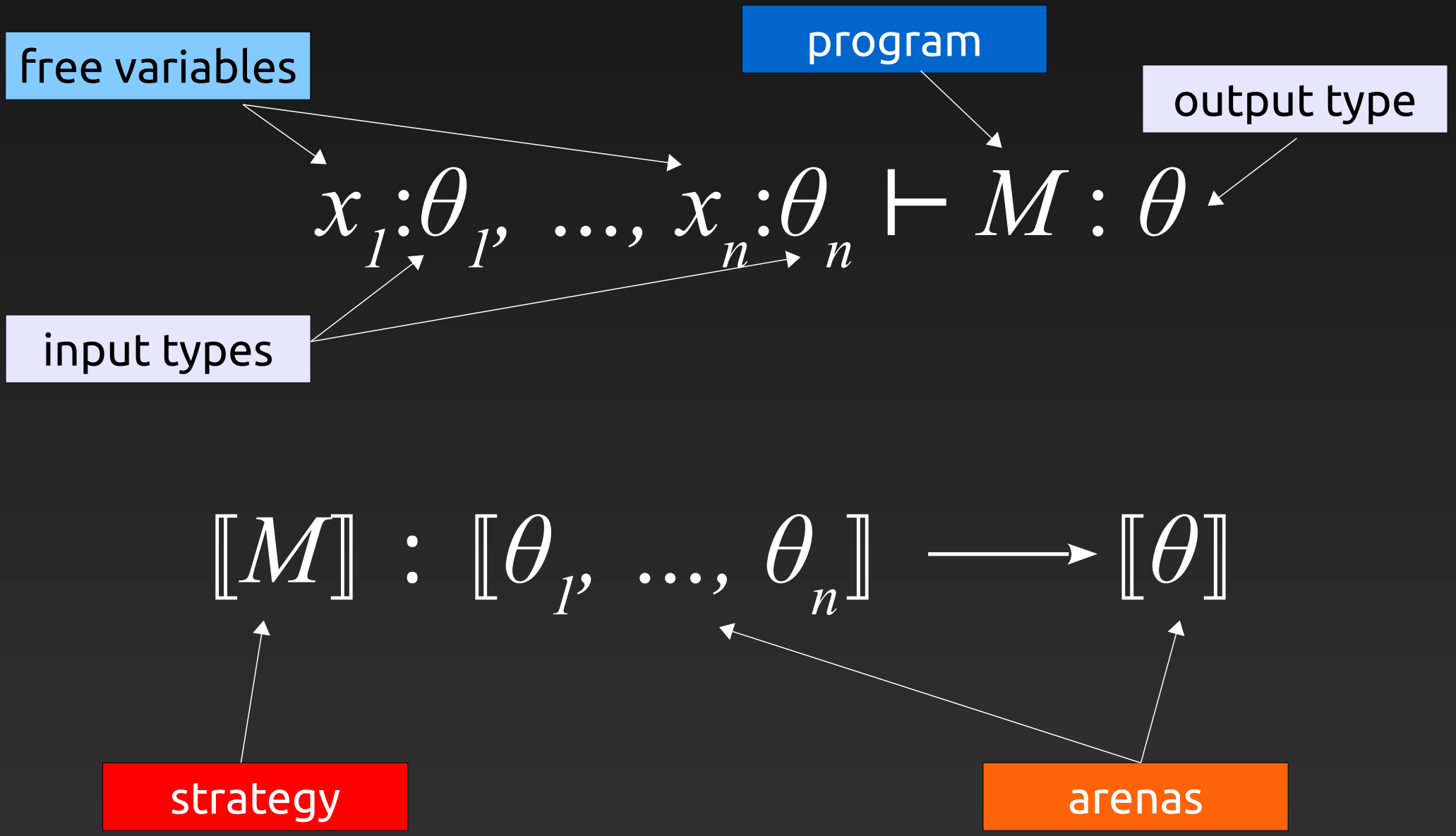
Programs = *strategies* for $P$

Families (i.e. *categories*) of games

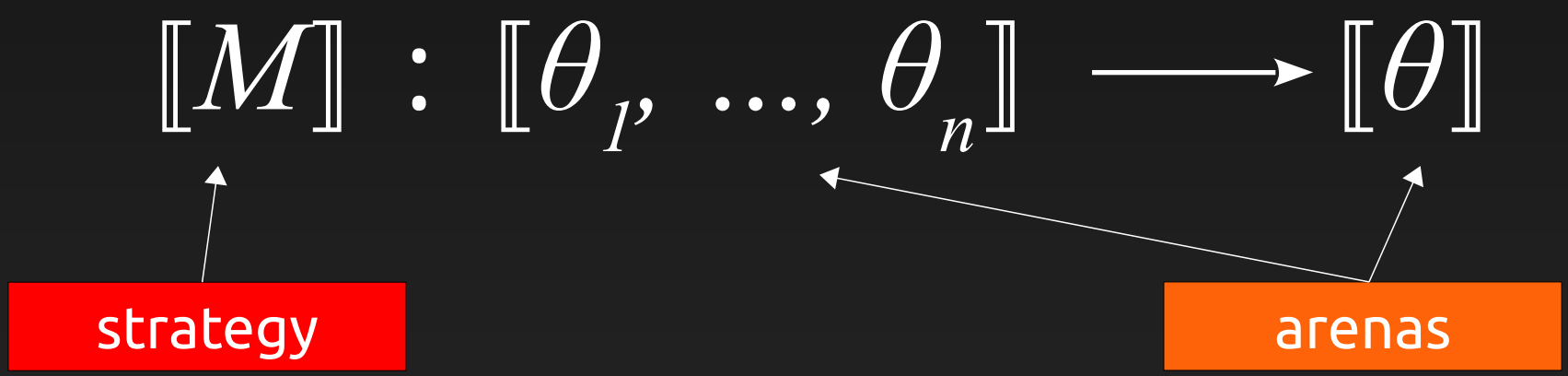
# Game infrastructure



# Game infrastructure



# Arenas of initial moves



# Arenas of initial moves

$$[\![M]\!] : [\![\theta_1, \ldots, \theta_n]\!] \longrightarrow [\![\theta]\!]$$

strategy

arenas

initial moves

$$[\![\text{void}]\!] = \{ * \}$$

$$[\![\text{int}]\!] = \{ 0, 1, -1, \ldots \}$$

$$[\![\mathcal{I}]\!] = \{ a, b, \ldots, n, \ldots \}$$

$$(+ \text{type} = \mathcal{I})$$

$$a, b, n, \ldots \in \mathcal{N}$$

$\mathcal{N}$ the set of *names*

# Games for IMJ programs

Program interactions are modelled by sequences of moves-with-store, written $m^{\Sigma}$, where:

stores are of the form:

$$\Sigma = \{ n \mapsto (\text{Emp}, \emptyset),\ a \mapsto (\text{Ptr}, \text{val} = a),\ p \mapsto (\text{Pt2}, \text{x} = 1, \text{y} = 0),$$
$$c \mapsto (\text{Ptr}', \emptyset),\ q \mapsto (\text{Pt2}', \emptyset),\ \ldots \}$$

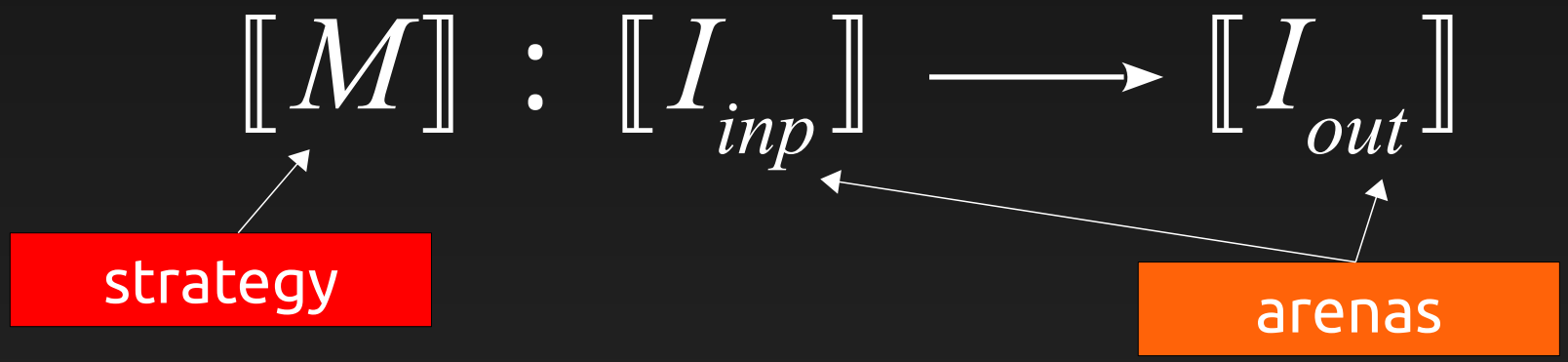
Ptr' : ( get : Ptr' → Ptr', set : Ptr' → Ptr' )
Pt2' : ( get : void → (int,int), set : (int,int) → void )

move $m$ is either:

- an initial move
- an object method call/return

call $c$.set($c'$),
ret $q$.get(5,4), …

# Plays, strategies




*Plays* : sequences of *moves-with-store*

call $n$.set(12) $^{(n \mapsto \text{IntCell, val=5})}$, ...

*Strategies* : sets of plays

- moves have polarities (*O*/*P*), which alternate
- *P* calls methods of *O*, and viceversa; dually for returns
- calls and returns obey the object interfaces
- strategies are closed wrt to *O*-subtyping

...

# Examples

$x$ : Var, $f$ : Fun $\vdash$ $f$.val ($x$.val) + 1 : int

Var : ( val : int ) Fun : ( val : int $\rightarrow$ int )

# Examples

$x$ : Var, $f$ : Fun $\vdash$ $f$.val ($x$.val) + 1 : int

Var : ( val : int ) Fun : ( val : int $\rightarrow$ int )

$O$ : $(x, f)^{(x.\text{val} = 4)}$

# Examples

$x$ : Var, $f$ : Fun $\vdash$ $f$.val ($x$.val) + 1 : int

Var : ( val : int ) Fun : ( val : int $\rightarrow$ int )

$O$ : $(x, f)^{(x.\text{val} = 4)}$

$P$ : call $f$.val(4)$^{(x.\text{val} = 4)}$

# Examples

$x$ : Var, $f$ : Fun $\vdash$ $f$.val ($x$.val) + 1 : int

Var : ( val : int ) Fun : ( val : int $\rightarrow$ int )

$O$ : $(x, f)^{(x.\text{val} = 4)}$

$P$ : call $f$.val(4)$^{(x.\text{val} = 4)}$

$O$ : ret $f$.val(50)$^{(x.\text{val} = 73)}$

# Examples

$x$ : Var, $f$ : Fun $\vdash$ $f$.val ($x$.val) + 1 : int

Var : ( val : int ) Fun : ( val : int $\rightarrow$ int )

$O$ : $(x, f)^{(x.\text{val} = 4)}$

$P$ : call $f$.val(4)$^{(x.\text{val} = 4)}$

$O$ : ret $f$.val(50)$^{(x.\text{val} = 73)}$

$P$ : 51$^{(x.\text{val} = 73)}$

$[\![ x$ : Var, $f$ : Fun $\vdash$ $f$.val ($x$.val) + 1 : int $]\!]$

$= \{ (x, f)^{(x.\text{val} = i)}$ call $f$.val($i$)$^{(x.\text{val} = i)}$ ret $f$.val($j$)$^{(x.\text{val} = i')}$ $(j+1)^{(x.\text{val} = i')}$ $\}$

# IMJ example: game semantics

$M_1$ : let $u$ = new($\mathrm{Var}_{Emp}$) in
new($\mathcal{M}_1$): Cell

$\mathcal{M}_1$ : get: $\lambda().\ u.\mathrm{val}$,
set: $\lambda y.\ u.\mathrm{val} := y$

$\Delta$ = Empty: $\emptyset$,
Cell: (get: void $\rightarrow$ Empty,
set: Empty $\rightarrow$ void),
$\mathrm{Var}_{Emp}$: (val: Empty),
$\mathrm{Var}_{Int}$: (val: int)

$$[\![M_1]\!] = \ *\ n^{\Sigma_0}\ (\ \mathrm{call}\ n.\mathrm{get}()^{\Sigma_0}\ \ \mathrm{ret}\ n.\mathrm{get}(\mathrm{nul})^{\Sigma_0}\ )^*\ \ \mathrm{call}\ n.\mathrm{set}(n_1)^{\Sigma_1}\ \ \mathrm{ret}\ n.\mathrm{set}()^{\Sigma_1}\ (\ \mathrm{call}\ n.\mathrm{get}()^{\Sigma_1}\ \ \mathrm{ret}\ n.\mathrm{get}(n_1)^{\Sigma_1}\ )^*\ \ \mathrm{call}\ n.\mathrm{set}(n_2)^{\Sigma_2}\ \ \mathrm{ret}\ n.\mathrm{set}()^{\Sigma_2}\ \ldots$$

O P O P

$$\Sigma_i = \{\ n \mapsto (\mathrm{Cell}, \emptyset)\ \} \cup \{\ n_j \mapsto (\mathrm{Empty}, \emptyset),\ 1 \leq j \leq i\ \}$$

# IMJ example: game semantics

$M_1$ : let $u$ = new($\mathrm{Var}_{Emp}$) in
new($\mathcal{M}_1$): Cell

$\mathcal{M}_1$ : get: $\lambda().\ u.\mathrm{val}$,
set: $\lambda y.\ u.\mathrm{val} := y$

$\Delta$ = Empty: $\emptyset$,
Cell: (get: void $\rightarrow$ Empty,
set: Empty $\rightarrow$ void),
$\mathrm{Var}_{Emp}$: (val: Empty),
$\mathrm{Var}_{Int}$: (val: int)

$$[\![M_1]\!] = * \; n^{\Sigma_0} \; (\; \mathrm{call}\; n.\mathrm{get}()^{\Sigma_0} \; \mathrm{ret}\; n.\mathrm{get}(\mathrm{nul})^{\Sigma_0} \;)^*$$

O P O P

$$\mathrm{call}\; n.\mathrm{set}(n_1)^{\Sigma_1} \; \mathrm{ret}\; n.\mathrm{set}()^{\Sigma_1}$$
$$(\; \mathrm{call}\; n.\mathrm{get}()^{\Sigma_1} \; \mathrm{ret}\; n.\mathrm{get}(n_1)^{\Sigma_1} \;)^*$$
$$\mathrm{call}\; n.\mathrm{set}(n_2)^{\Sigma_2} \; \mathrm{ret}\; n.\mathrm{set}()^{\Sigma_2} \; \ldots$$

$$\Sigma_i = \{\; n \mapsto (\mathrm{Cell}, \emptyset) \;\} \cup \{\; n_j \mapsto (\mathrm{Empty}, \emptyset),\ 1 \leq j \leq i \;\}$$

# IMJ example: game semantics

$M_1$ : let $u$ = new($\mathrm{Var}_{Emp}$) in
new($\mathcal{M}_1$): Cell

$\mathcal{M}_1$ : get: $\lambda().\ u.\mathrm{val}$,
set: $\lambda y.\ u.\mathrm{val} := y$

$\Delta$ = Empty: $\emptyset$,
Cell: (get: void $\rightarrow$ Empty,
set: Empty $\rightarrow$ void),
$\mathrm{Var}_{Emp}$: (val: Empty),
$\mathrm{Var}_{Int}$: (val: int)

$$[\![M_1]\!] = * \; n^{\Sigma_0} \; (\; \underset{O}{\text{call } n.\text{get}()^{\Sigma_0}} \; \underset{P}{\text{ret } n.\text{get}(\text{nul})^{\Sigma_0}} \;)^* \; \underset{O}{\text{call } n.\text{set}(n_1)^{\Sigma_1}} \; \underset{P}{\text{ret } n.\text{set}()^{\Sigma_1}} \; (\; \text{call } n.\text{get}()^{\Sigma_1} \; \text{ret } n.\text{get}(n_1)^{\Sigma_1} \;)^* \; \text{call } n.\text{set}(n_2)^{\Sigma_2} \; \text{ret } n.\text{set}()^{\Sigma_2} \; \ldots$$

$$\Sigma_i = \{\; n \mapsto (\text{Cell}, \emptyset) \;\} \cup \{\; n_j \mapsto (\text{Empty}, \emptyset),\ 1 \leq j \leq i \;\}$$

# IMJ example: game semantics

$M_1$ : let $u$ = new($\mathrm{Var}_{Emp}$) in
new($\mathcal{M}_1$): Cell

$\mathcal{M}_1$ : get: $\lambda().\ u.\mathrm{val}$,
set: $\lambda y.\ u.\mathrm{val} := y$

$\Delta$ = Empty: $\emptyset$,
Cell: (get: void → Empty,
set: Empty → void),
$\mathrm{Var}_{Emp}$: (val: Empty),
$\mathrm{Var}_{Int}$: (val: int)

$$[\![M_1]\!] = * \; n^{\Sigma_0} \; (\; \underset{O}{\text{call } n.\text{get}()^{\Sigma_0}} \; \underset{P}{\text{ret } n.\text{get}(\text{nul})^{\Sigma_0}} \;)^*$$
$$\underset{O}{\text{call } n.\text{set}(n_1)^{\Sigma_1}} \; \underset{P}{\text{ret } n.\text{set}()^{\Sigma_1}}$$
$$(\; \underset{O}{\text{call } n.\text{get}()^{\Sigma_1}} \; \underset{P}{\text{ret } n.\text{get}(n_1)^{\Sigma_1}} \;)^*$$
$$\underset{O}{\text{call } n.\text{set}(n_2)^{\Sigma_2}} \; \underset{P}{\text{ret } n.\text{set}()^{\Sigma_2}} \; \ldots$$

$$\Sigma_i = \{\; n \mapsto (\text{Cell}, \emptyset) \;\} \cup \{\; n_j \mapsto (\text{Empty}, \emptyset),\ 1 \le j \le i \;\}$$

# IMJ example: game semantics

$M_1$ : let $u$ = new($\mathrm{Var}_{Emp}$) in
new($\mathcal{M}_1$): Cell

$\mathcal{M}_1$ : get: $\lambda().\ u.\mathrm{val}$,
set: $\lambda y.\ u.\mathrm{val} := y$

$\Delta$ = Empty: $\emptyset$,
Cell: (get: void $\rightarrow$ Empty,
set: Empty $\rightarrow$ void),
$\mathrm{Var}_{Emp}$: (val: Empty),
$\mathrm{Var}_{Int}$: (val: int)

$$[\![M_1]\!] = * \; n^{\Sigma_0} \; (\; \underset{O}{\text{call } n.\text{get}()^{\Sigma_0}} \; \underset{P}{\text{ret } n.\text{get}(\text{nul})^{\Sigma_0}} \;)^*$$
$$\underset{O}{\text{call } n.\text{set}(n_1)^{\Sigma_1}} \; \underset{P}{\text{ret } n.\text{set}()^{\Sigma_1}}$$
$$(\; \text{call } n.\text{get}()^{\Sigma_1} \; \text{ret } n.\text{get}(n_1)^{\Sigma_1} \;)^*$$
$$\text{call } n.\text{set}(n_2)^{\Sigma_2} \; \text{ret } n.\text{set}()^{\Sigma_2} \; \ldots$$

$$\Sigma_i = \{\; n \mapsto (\text{Cell}, \emptyset) \;\} \cup \{\; n_j \mapsto (\text{Empty}, \emptyset),\ 1 \leq j \leq i \;\}$$

# IMJ example: game semantics

$M_2$ : let $b$ = new($\mathrm{Var}_{Int}$) in
let $u_1$ = new($\mathrm{Var}_{Emp}$) in
let $u_2$ = new($\mathrm{Var}_{Emp}$) in
new($\mathcal{M}_2$) : Cell

$\mathcal{M}_2$ : get: $\lambda()$. if $b$.val
then $b$.val := 0; $u_1$.val
else $b$.val := 1; $u_2$.val,
set: $\lambda y$. $u_1$.val := $y$;
$u_2$.val := $y$

**O** **P** **O** **P**

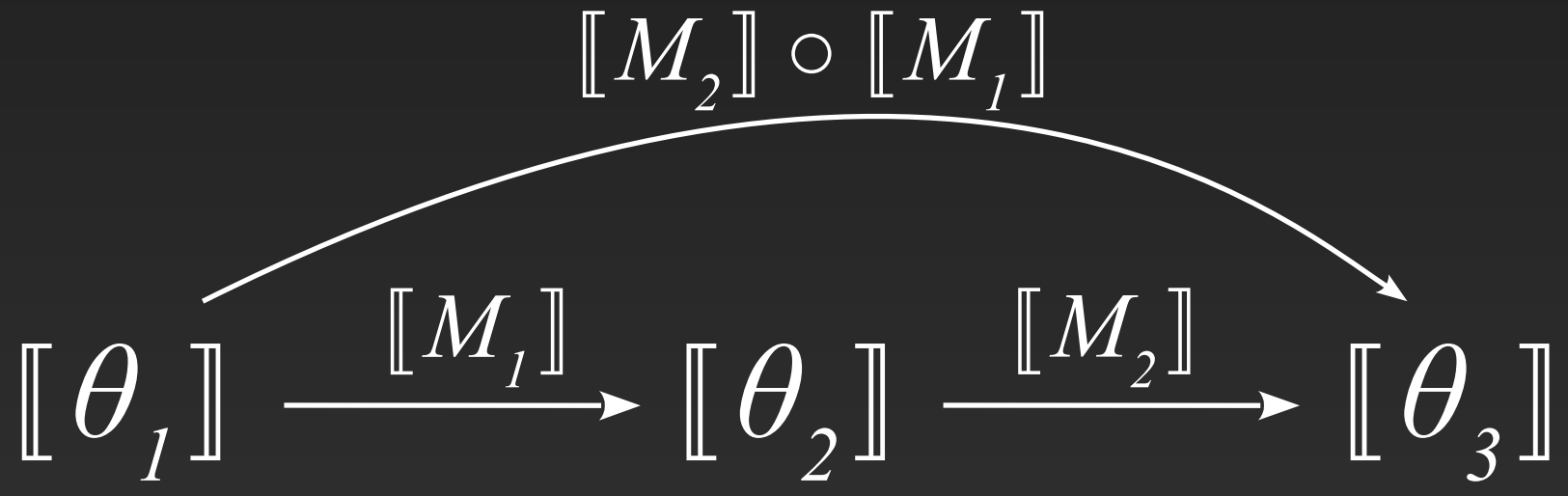
$[\![M_1]\!]$ = $*$ $n^{\Sigma_0}$ ( call $n$.get()$^{\Sigma_0}$ ret $n$.get(nul)$^{\Sigma_0}$ )*
call $n$.set($n_1$)$^{\Sigma_1}$ ret $n$.set()$^{\Sigma_1}$
( call $n$.get()$^{\Sigma_1}$ ret $n$.get($n_1$)$^{\Sigma_1}$ )*
call $n$.set($n_2$)$^{\Sigma_2}$ ret $n$.set()$^{\Sigma_2}$ ... $= [\![M_2]\!]$

$\Sigma_i = \{ n \mapsto (\mathrm{Cell}, \emptyset) \} \cup \{ n_j \mapsto (\mathrm{Empty}, \emptyset), 1 \leq j \leq i \}$

# Composition

Compound programs translated compositionally

- Strategy composition: play one strategy against the other



# Composition issues

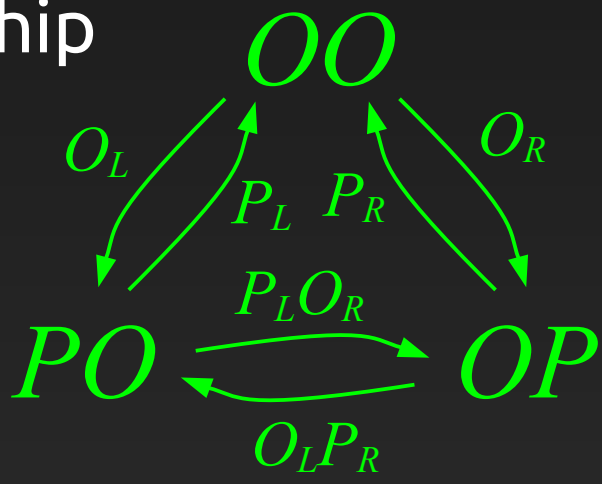
Playing strategies against each other requires to:

- Dynamically determine object ownership
- Extract which moves are interacting
- Match & hide interacting *O*/*P* moves

# Composition issues

Playing strategies against each other requires to:

- Dynamically determine object ownership
- Extract which moves are interacting
- Match & hide interacting *O*/*P* moves




$$[\![\theta_1]\!] \xrightarrow{[\![M_1]\!]} [\![\theta_2]\!] \xrightarrow{[\![M_2]\!]} [\![\theta_3]\!]$$

# Full abstraction for IMJ

*Lemma.* The game model is sound

*Lemma.* Every finitary strategy is IMJ-definable

*Theorem.* The game model is fully abstract

$$P \sqsubseteq P' \iff [\![P]\!] \subseteq [\![P']\!]$$

# Related work

Domain models

Environmental bisimulations

Trace models

*TAOOP'94*

**Two Semantic Models of Object-Oriented Languages***

Samuel N. Kamin
Uday S. Reddy
University of Illinois at Urbana-Char

October 4, 1993

**Abstract**

We present and compare two models of object-oriented lang
*closure model* because it uses closures to encapsulate side effects
makes the operations on an object a part of that object. It is s
framework is adequate to explain classes, instantiation, and
Simula as well as SMALLTALK–80. The second we call the *dat*
mimics the implementations of data structure languages like
by records of instance variables, while keeping the operations
objects themselves. This yields a model which is very simple,
the models are presented by way of a sequence of languages,
with SMALLTALK–80-style inheritance. The mathematical re
then discussed and it is shown that the models give equivalent
this discussion that more appropriate names for the two mod
*model* and the *self-application model*.

## 1 Introduction

Object-oriented languages, such as SMALLTALK–80[1] [GR83],
attention. However, the term "object-oriented" does not seem
meaning. It is sometimes used to refer to the presence of *data ob*
to the *coupling* of data with operations, sometimes to record *sub*
of *class inheritance*, and sometimes to the specific notion of
involves a kind of "dynamic binding". The first of these notion
has long been used in the functional programming communit
whenever they were necessary [ASS85, KL85]. These are so
*closures*. A closure is essentially a function or a data structure
local bindings to values or storage locations. In describing th
languages, it seems natural that such a notion of closure should

*To appear in Gunter, C. and Mitchell, J. C. (eds) *Theoretical aspects o*
Press, 1993.

[1] "SMALLTALK–80" is a trademark of ParcPlace Systems. We use here
different capitalization) as an abstraction of SMALLTALK–80.



*FSSJava'99*

**Dynamic Denotational Semantics of Java**

Jim Alves-Foss and Fong Shing Lam

Center for Secure and Dependable Software, Department of Computer Science,
University of Idaho, Moscow ID 83844-1010, USA

**Abstract.** This chapter presents a dynamic denotational semantics of the Java programming language. This semantics covers almost the full range of the base language, excluding only concurrency and the APIs. A discussion of these limitations is provided in the final section of the chapter.

The abstract syntax described in Chapter 1 tells us how to construct a grammatically correct program. Every syntactically correct program describes an *environment* that provides all the information about what to do during program execution. The semantics presented in this chapter, formalizes the definition of Java program beahior as defined in the *Java Language Specification* (JLS) [1]. We describe the Java environment in Section 1. Each executing program is associated with a *store* that is a repository for all instance values during program execution. The Java store is described in Section 2. Executing a Java program begins with executing the command in the static method "main" in the given class definition. Therefore, the result of a program depends on the semantics of commands and the expressions in the commands. We shall introduce a denotational semantics of these commands and expressions in Sections 3 and 4. Throughout these semantics, we concurrently define two sets of semantics, a *dynamic* and a *static* semantics, to respectively represent the execution and definitional denotations of the programs.

## 1 Environment

An *environment* is the information center for the execution engine and is at the heart of these semantics. Our environment is a semantic domain that has two components, the dynamic and static semantics. The *dynamic* aspect of the environment contains the traditional environmental information related to variables, their types and locations in the store (as in Stoy's classical book on denotational semantics [4]. It also contains control flow information for exceptions and breaks. The *static* aspect of the environment contains information related to all of the classes used by the program. This information includes the class members, types, initialization functions, super class and implemented interfaces. The static part of the environment is determined by evaluating the input files and then is used as an input parameter to the denotation of the main method of the invoked class.

# Related work

Domain models

Environmental bisimulations

Trace models

*TAOOP'94*

**Two Semantic Models of Object-Oriented Languages***

Samuel N. Kamin
Uday S. Reddy
University of Illinois at Urbana-Char

October 4, 1993

**Abstract**

We present and compare two models of object-oriented lang
closure model because it uses closures to encapsulate side effects

*FSSJava'99*

**Dynamic Denotational Semantics of Java**

Jim Alves-Foss and Fong Shing Lam

Center for Secure and Dependable Software, Department of Computer Science,
University of Idaho, Moscow ID 83844-1010, USA

its a dynamic denotational semantics of
. This semantics covers almost the full
luding only concurrency and the APIs.
is provided in the final section of the

Chapter 1 tells us how to construct a gram-
ntactically correct program describes an ex-
rmation about what to do during program
in this chapter, formalizes the definition of
the *Java Language Specification* (JLS) [1].
n Section 1. Each executing program is as-
itory for all instance values during program
ed in Section 2. Executing a Java program
l in the static method "main" in the given
lt of a program depends on the semantics
n the commands. We shall introduce a de-
nands and expressions in Sections 3 and 4.
ncurrently define two sets of semantics, a
espectively represent the execution and def-
s.

enter for the execution engine and is at the
onment is a semantic domain that has two
semantics. The *dynamic* aspect of the envi-
ironmental information related to variables,
(as in Stoy's classical book on denotational
l flow information for exceptions and breaks.
t contains information related to all of the
ormation includes the class members, types,
nd implemented interfaces. The static part
evaluating the input files and then is used
ion of the main method of the invoked class.

*FOOL/WOOD'07*

**Reasoning about Class Behavior**

Vasileios Koutavas
Northeastern University
vkoutav@ccs.neu.edu

Mitchell Wand
Northeastern University
wand@ccs.neu.edu

**Abstract**

We present a sound and complete method for reasoning about contextual equivalence of classes in an imperative subset of Java. To the extent of our knowledge this is the first such method for a language with unrestricted inheritance and downcasting, where the context can arbitrarily extend classes to distinguish otherwise equivalent implementations. Similar reasoning techniques for class-based languages [1, 12] don't consider inheritance at all, or forbid the context from extending related classes. Other techniques that do consider inheritance [3] study whole-program equivalence. Using our technique we were able to prove equivalences in examples that make use of public, private, and protected interfaces of classes, imperative fields, and invocations of callbacks—a higher-order feature, where other methods admit limitations [20, 22].

We apply our technique multiple times to consecutive extensions of a base language, adding first inheritance and then downcasting. We demonstrate that these extensions only incrementally affect our technique and our main theorem, and argue that the effect captures the intuition of each extension.

Furthermore we give the definition of an equivalence weaker than contextual equivalence and show how our technique can be adapted to reason about it.

**1. Introduction**

The class is a facility to divide programs into small units that encode different parts of the entire program behavior. This makes classes attractive for reuse and re-implementation. But changing the implementation of a class that is being used in a number of programs comes with the responsibility that the new implementation will not alter the behavior of these programs.

The effect that a change in the implementation of a class has on the behavior of a program that uses it depends greatly on the ways that the program interacts with the class. In a Java-like language (and in the absence of reflection) the surrounding program can interact directly with the class by creating new instances, invoking its public methods, and changing the state of its public fields. It can also interact in a more indirect way with the class. It can define subclasses that extend the original class and instantiate objects, invoke methods, and change the state of fields of these classes.

To formalize the notion of equivalent implementations of classes we adapt the standard notion of contextual equivalence between expressions from functional languages [19] to an equivalence between classes in class-based languages: classes $C$ and $C'$ are contextually equivalent, if and only if, for all *class-table contexts* $\mathcal{CT}[\,]$, expressions $e$, and the empty store $\emptyset$, the program configurations $(\mathcal{CT}[C], \emptyset, e)$ and $(\mathcal{CT}[C'], \emptyset, e)$ have the same operational behavior.

Using this definition directly for proving the equivalence of two sufficiently different implementations of a class is not possible. This is because of the quantification over all class table contexts, but also because it is not strong enough to support an inductive proof which would require us to consider not just equal, but also related stores. CIU theorems [16] ease the quantification over contexts by considering only the evaluation contexts, but they similarly are not strong enough in general to support an inductive proof. Moreover CIU theorems have not been applied to class-based languages.

Another way of reasoning about the behavior of class implementations is by using denotational methods (see [6, 13]). Denotations are usually compositional in the sense that they give the meaning of program fragments without the quantification over contexts. Nevertheless the usual denotational methods distinguish equivalent class implementations that have different local store behaviors. For example the two implementations of a Cell class in Figure 1 would have different denotations because they have different fields. Such equivalences can be dealt with by methods that build logical relations of denotations [5], or exploit properties of some programs, such as ownership confinement [3]. However, these methods are still not complete with respect to contextual equivalence.

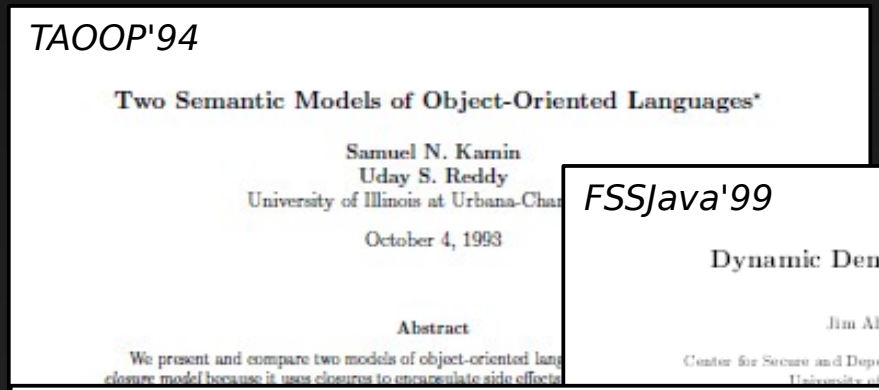
A more natural way to reason about the behavior of two program fragments is by using bisimulations. Bisimulations were introduced by Hennessy and Milner [9] for reasoning about the behavior of concurrent programs. They were applied in sequential calculi by Abramsky [2], and Howe [10] gave a way of proving that they are a congruence. Sumii and Pierce later gave a big-step bisimulation proof technique which is sound and complete with respect to contextual equivalence in a language with dynamic sealing [21] and in a language with recursive and polymorphic types [22]. Their key innovation was to split the sets into parts, and associate each part with

# Related work

Domain models

Environmental bisimulations

Trace models



*TAOOP'94*

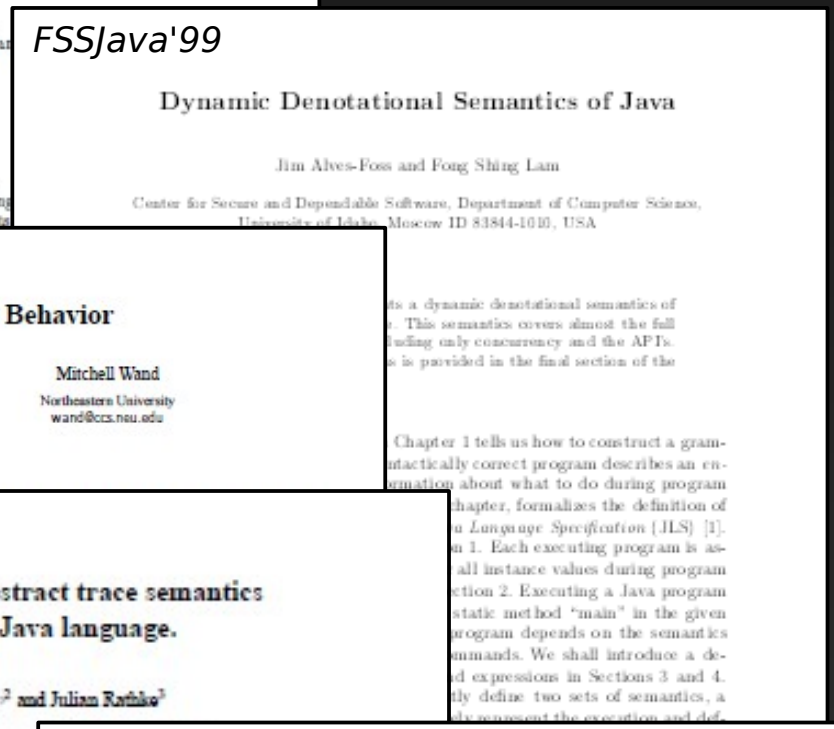
**Two Semantic Models of Object-Oriented Languages***

Samuel N. Kamin
Uday S. Reddy
University of Illinois at Urbana-Char

October 4, 1993

**Abstract**

We present and compare two models of object-oriented lang
closure model because it uses closures to encapsulate side effects



*FSSJava'99*

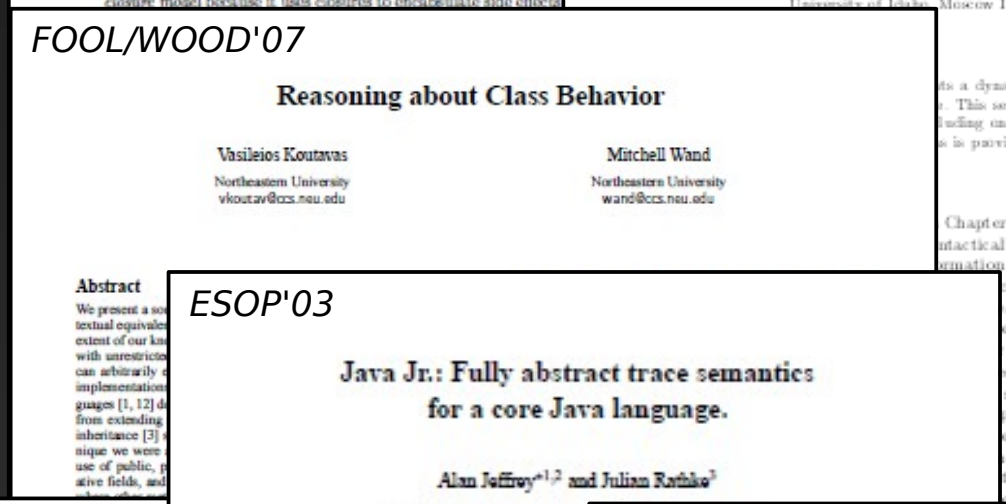
**Dynamic Denotational Semantics of Java**

Jim Alves-Foss and Fong Shing Lam

Center for Secure and Dependable Software, Department of Computer Science,
University of Idaho, Moscow ID 83844-1010, USA

its a dynamic denotational semantics of
. This semantics covers almost the full
luding only concurrency and the APTs.
is provided in the final section of the

Chapter 1 tells us how to construct a gram-
ntactically correct program describes an ex-
rmation about what to do during program
hapter, formalizes the definition of
a Language Specification (JLS) [1].
n 1. Each executing program is as-
all instance values during program
ction 2. Executing a Java program
static method "main" in the given
program depends on the semantics
mmands. We shall introduce a de-
d expressions in Sections 3 and 4.
fly define two sets of semantics, a



*FOOL/WOOD'07*

**Reasoning about Class Behavior**

Vasileios Koutavas
Northeastern University
vkoutav@ccs.neu.edu

Mitchell Wand
Northeastern University
wand@ccs.neu.edu

**Abstract**

We present a se
textual equivale
extent of our kn
with unrestricte
can arbitrarily
implementation
guages [1, 12] d
from extending
inheritance [3]
nique we were
use of public,
ative fields, and



*ESOP'03*

**Java Jr.: Fully abstract trace semantics for a core Java language.**

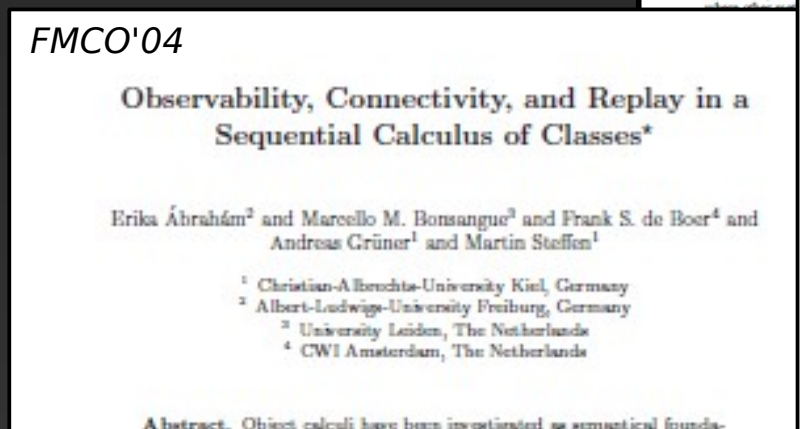
Alan Jeffrey*[1,2] and Julian Rathke[3]

[1] Bell Labs, Lucent Techno
[2] DePaul University,
[3] University of Sus

**Abstract.** We introduce an expressive ye
guage, Java Jr., and provide it with a form
of observable actions which represent int
detailed example based on the Observer
itive character of the semantic model. We
alence is fully-abstract with respect to a
object systems. This is the first such res
with inheritance.

**1 Introduction**

Operational semantics as a modelling tool fo



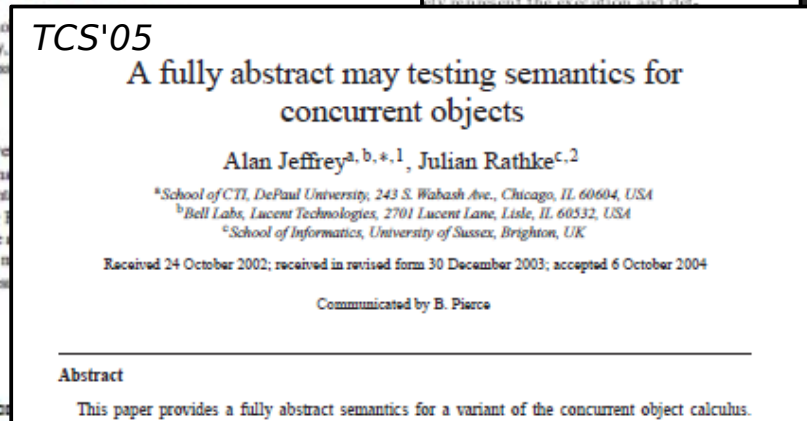
*FMCO'04*

**Observability, Connectivity, and Replay in a Sequential Calculus of Classes***

Erika Ábrahám[2] and Marcello M. Bonsangue[3] and Frank S. de Boer[4] and Andreas Grüner[1] and Martin Steffen[1]

[1] Christian-Albrechts-University Kiel, Germany
[2] Albert-Ludwigs-University Freiburg, Germany
[3] University Leiden, The Netherlands
[4] CWI Amsterdam, The Netherlands

**Abstract.** Object calculi have been investigated as semantical founda-



*TCS'05*

**A fully abstract may testing semantics for concurrent objects**

Alan Jeffrey[a, b, *, 1], Julian Rathke[c, 2]

[a] School of CTI, DePaul University, 243 S. Wabash Ave., Chicago, IL 60604, USA
[b] Bell Labs, Lucent Technologies, 2701 Lucent Lane, Lisle, IL 60532, USA
[c] School of Informatics, University of Sussex, Brighton, UK

Received 24 October 2002; received in revised form 30 December 2003; accepted 6 October 2004

Communicated by B. Pierce

**Abstract**

This paper provides a fully abstract semantics for a variant of the concurrent object calculus.

# Further on

## Program analysis for IMJ

- Algorithmic representations
- Automata over infinite alphabets
- Equivalence tests (complete fragment)

## Further effects & analyses

- Exceptions (cf. FoSSaCS'14)
- Concurrency (cf. Jeffrey & Rathke '05, Laird '06)
- Model checking

# Further on

thanks!

## Program analysis for IMJ

- Algorithmic representations
- Automata over infinite alphabets
- Equivalence tests (complete fragment)

## Further effects & analyses

- Exceptions (cf. FoSSaCS'14)
- Concurrency (cf. Jeffrey & Rathke '05, Laird '06)
- Model checking