GAME SEMANTICS FOR INTERFACE MIDDLEWEIGHT JAVA

Andrzej S. Murawski
WARWICK

Steven J. Ramsay
OXFORD

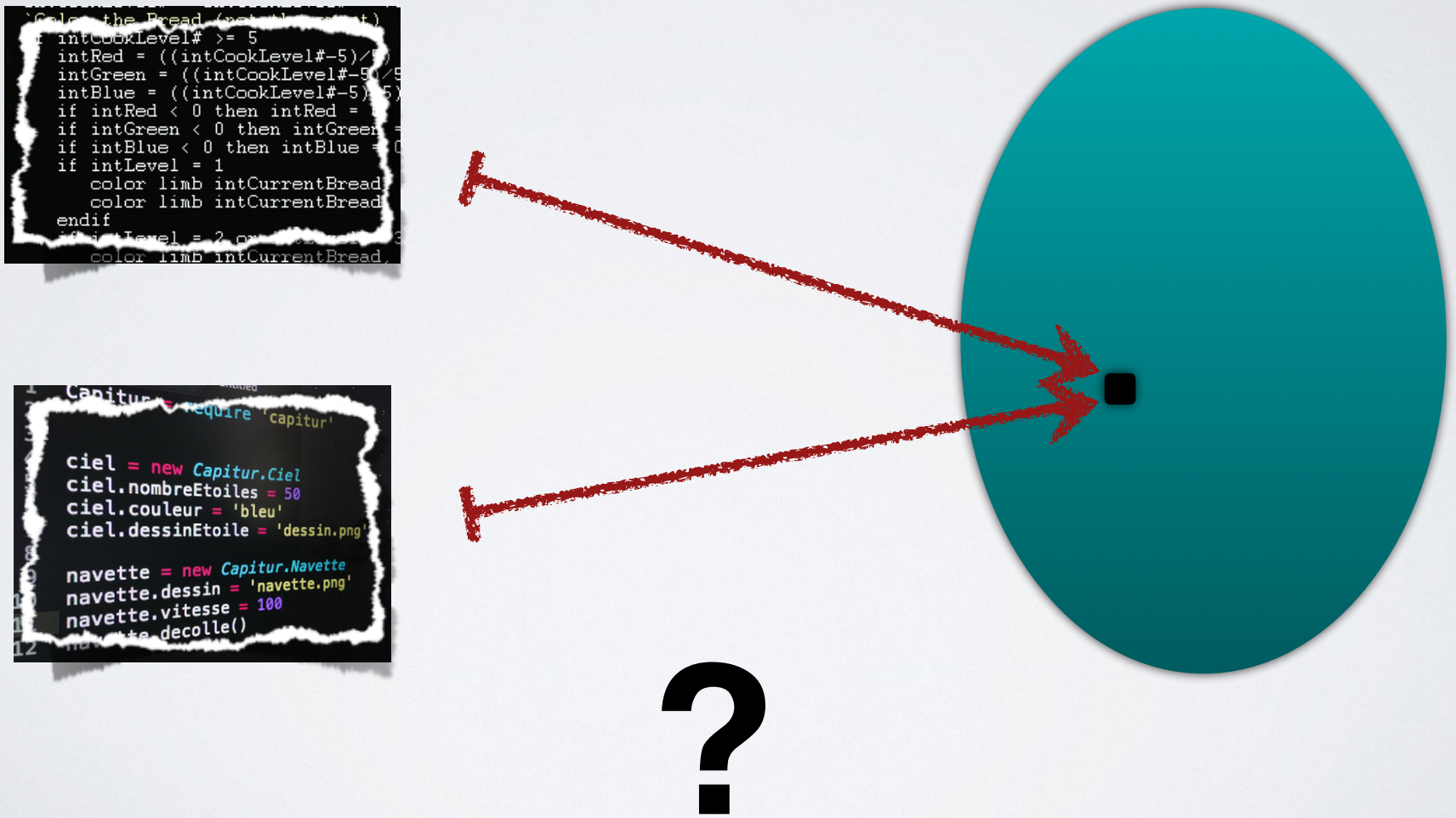
Nikos Tzevelekos
QUEEN MARY LONDON

WHAT IS THIS TALK ABOUT?

a fully abstract denotational model
for a core fragment of Java

classification of decidable cases
for contextual equivalence

CAPTURING PROGRAM BEHAVIOUR



FULL ABSTRACTION

$$[\![M_1]\!] = [\![M_2]\!]$$

if and only if

$$M_1 \cong M_2$$

Game semantics: from PCF to ML

Full Abstraction for PCF (early 90's)

Games for variants of Idealized Algol

- Non-determinism, exceptions, probability, concurrency, polymorphism, …

Nominal game semantics (2004-)

- *Use names for dynamic resource generation*

  → fragments of ML, CML, C, now Java

The need for names

References as *pairs:*

$\texttt{ref int} = (\texttt{unit} \to \texttt{int}) \times (\texttt{int} \to \texttt{unit})$

$\Longrightarrow \; (1 \to Z) \times (Z \to 1)$

- Theoretically attractive
- but: $\texttt{mkvar}(R, H)$, all R, H (*bad variables*)

The need for names

References as *pairs:*

$\texttt{ref int} = (\texttt{unit} \to \texttt{int}) \times (\texttt{int} \to \texttt{unit})$

$\Longrightarrow \; (1 \to Z) \times (Z \to 1)$

- Theoretically attractive
- but: $\texttt{mkvar}(R, H)$, all R, H (*bad variables*)

References as *names:*

$\texttt{ref int} = \text{base type}$

$\Longrightarrow \; \mathbb{A}$ (reference names)

- Notion of *resource (name):*
  - atomic values
  - infinitely many
  - comparable for equality

GOOD-VARIABLE SOLUTIONS

ref(int)

Andrzej S. Murawsk… Full Abstraction for Reduced ML. FOSSACS 2009: 32-47

ref(ref(θ))
ref(θ1 → θ2)

Andrzej S. Murawski, Nikos Tzevelekos: Game Semantics for Good General References. LICS 2011: 75-84

exceptions

Andrzej S. Murawski, Nikos Tzevelekos: Game Semantics for Nominal Exceptions. FoSSaCS 2014: 164-179

objects

Andrzej S. Murawski, Nikos Tzevelekos: Game semantics for interface middleweight Java. POPL 2014: 517-528

Interface Middleweight Java (IMJ)

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

Interface definitions

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

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

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

Method implementations

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

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

- Objects, inheritance, casting, interfaces

IMJ example*

M_1 : let u = new(Var_{Emp}) in
new(M_1) : Cell

M_1 : get: $\lambda()$. u.val,
set: λy. u.val := y

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

* Koutavas & Wand, 2007

IMJ example*

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

M_1 : let u = new(Var_{Emp}) in
new(M_1) : Cell

M_1 : get: $\lambda()$. u.val,
set: λy. u.val := y

M_2 : let b = new(Var_{Int}) in
let u_1 = new(Var_{Emp}) in
let u_2 = new(Var_{Emp}) in
new(M_2) : Cell

M_2 : get: $\lambda()$. if b.val
then b.val := 0; u_1.val
else b.val := 1; u_2.val,
set: λy. u_1.val := y;
u_2.val := y

* Koutavas & Wand, 2007

Game Semantics

Computation is modelled as a 2-player game between:

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

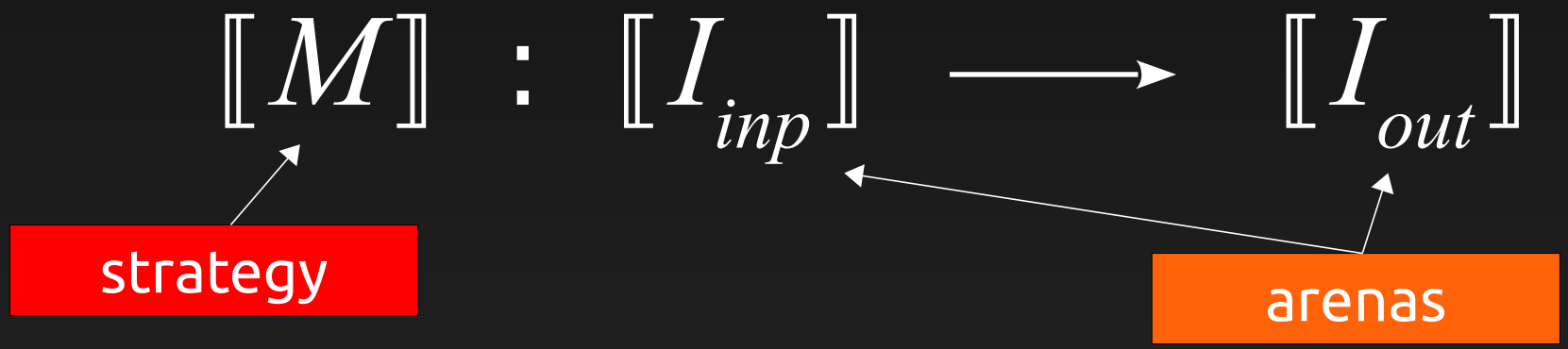
Qualitative games (≠ Game Theory)

Computations = *plays* of a specified game

Programs = *strategies* for P

Strategy composition → *categories* of games

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*-name subtyping

...

IMJ example: game semantics

M_1 : let u = new(Var_{Emp}) in
new(M_1) : Cell

M_1 : get: $\lambda().\ u.\mathrm{val}$,
set: $\lambda y.\ u.\mathrm{val} := y$

Δ = Empty: $\emptyset$,
Cell: (get: void → Empty,
set: Empty → void),
Var_{Emp}: (val: Empty),
Var_{Int}: (val: int)

$$[\![M_1]\!] = \quad \begin{matrix} O & P \\ * & n^{\Sigma_0} \end{matrix} \quad \begin{matrix} O & P \\ (\ \mathrm{call}\ n.\mathrm{get}()^{\Sigma_0} & \mathrm{ret}\ n.\mathrm{get}(\mathrm{nul})^{\Sigma_0}\ )^* \\ \mathrm{call}\ n.\mathrm{set}(n_1)^{\Sigma_1} & \mathrm{ret}\ n.\mathrm{set}()^{\Sigma_1} \\ (\ \mathrm{call}\ n.\mathrm{get}()^{\Sigma_1} & \mathrm{ret}\ n.\mathrm{get}(n_1)^{\Sigma_1}\ )^* \\ \mathrm{call}\ n.\mathrm{set}(n_2)^{\Sigma_2} & \mathrm{ret}\ n.\mathrm{set}()^{\Sigma_2} \ldots \end{matrix}$$

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

IMJ example: game semantics

M_1 : let u = new(Var_{Emp}) in
new(M_1) : Cell

M_1 : get: $\lambda().\ u.\mathrm{val}$,
set: $\lambda y.\ u.\mathrm{val} := y$

Δ = Empty: $\emptyset$,
Cell: (get: void → Empty,
set: Empty → void),
Var_{Emp}: (val: Empty),
Var_{Int}: (val: int)

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

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

$\mathrm{call}\; n.\mathrm{set}(n_1)^{\Sigma_1} \; \mathrm{ret}\; n.\mathrm{set}()^{\Sigma_1}$

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

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

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

IMJ example: game semantics

M_1 : let u = new(Var_{Emp}) in
new(M_1) : Cell

M_1 : get: $\lambda().\ u.\mathrm{val}$,
set: $\lambda y.\ u.\mathrm{val} := y$

Δ = Empty: $\emptyset$,
Cell: (get: void → Empty,
set: Empty → void),
Var_{Emp}: (val: Empty),
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()$^{\Sigma_2}$ ...

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

IMJ example: game semantics

M_1 : let u = new(Var_{Emp}) in
new(M_1) : Cell

M_1 : get: $\lambda().\ u.\mathrm{val}$,
set: $\lambda y.\ u.\mathrm{val} := y$

Δ = Empty: $\emptyset$,
Cell: (get: void → Empty,
set: Empty → void),
Var_{Emp}: (val: Empty),
Var_{Int}: (val: int)

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

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

IMJ example: game semantics

M_1 : let u = new(Var_{Emp}) in
new(M_1) : Cell

M_1 : get: $\lambda().\ u.\mathrm{val}$,
set: $\lambda y.\ u.\mathrm{val} := y$

Δ = Empty: $\emptyset$,
Cell: (get: void → Empty,
set: Empty → void),
Var_{Emp}: (val: Empty),
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_2 : let b = new(Var_{Int}) in
let u_1 = new(Var_{Emp}) in
let u_2 = new(Var_{Emp}) in
new(M_2) : Cell

M_2 : get: $\lambda()$. if b.val
then b.val := 0; u_1.val
else b.val := 1; u_2.val,
set: λy. u_1.val := y;
u_2.val := y

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

$[\![M_1]\!]$ = $*$ n^{Σ_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()$^{\Sigma_2}$ ... $= [\![M_2]\!]$

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

Full abstraction for IMJ

Lemma. The game model is sound

Lemma. Every finitary strategy is IMJ-definable

Theorem. The game model is fully abstract

$$P \sqsubseteq P' \quad \Leftrightarrow \quad [\![P]\!] \subseteq [\![P']\!]$$

Related work on objects

Domain 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.

1

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 on objects

Domain models

Environmental bisimulations

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,
and 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.

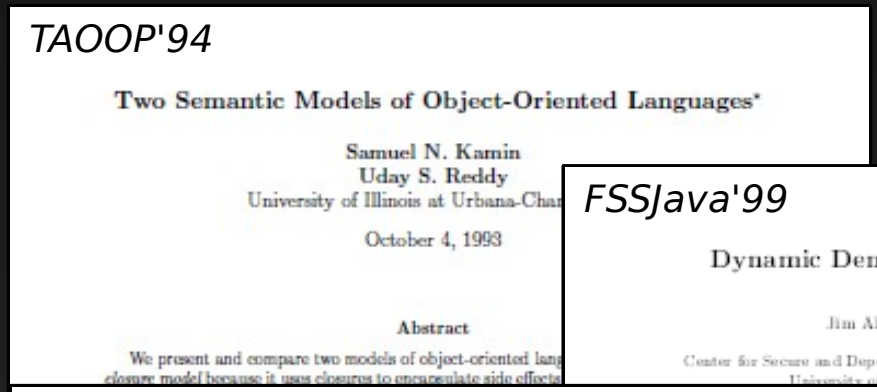
A more natural way to reason about the behavior of two program fragments is by using bisimulations. Bisimulations were introduced by Hennessy and Milner [9] for reasoning about the behavior of concurrent programs. They were applied in sequential calculi by Abramsky [2], and Howe [10] gave a way of proving 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 on objects

Domain models

Environmental bisimulations

Trace models



TAOOP'94

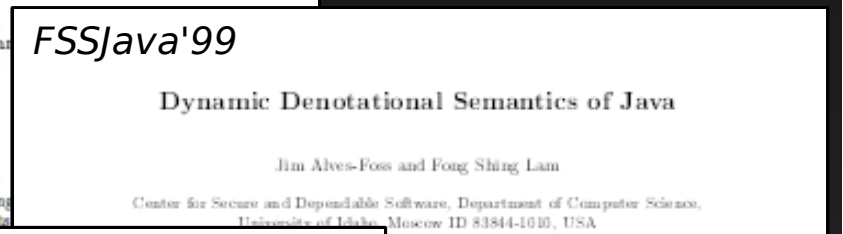
Two Semantic Models of Object-Oriented Languages*

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



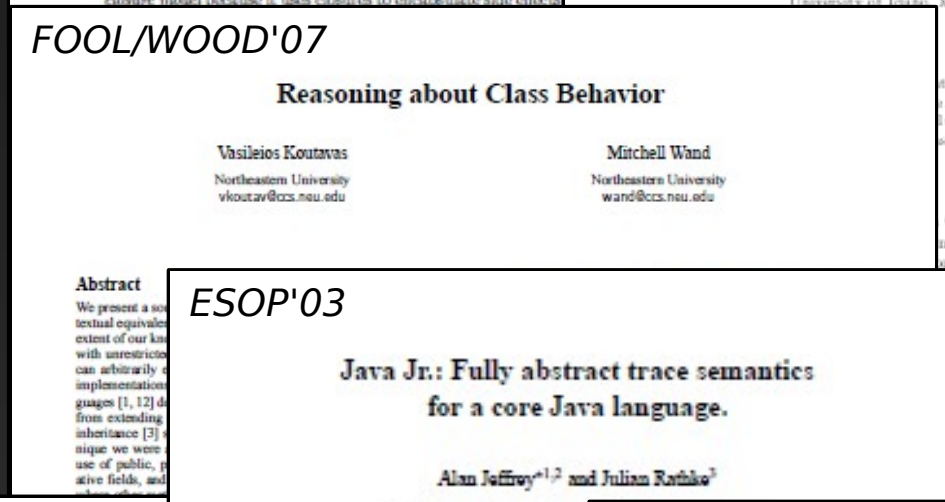
FSSJava'99

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

ts 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



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 so
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

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
rogram depends on the semantics
mmands. We shall introduce a de-
d expressions in Sections 3 and 4.
fly define two sets of semantics, a

ESOP'03

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

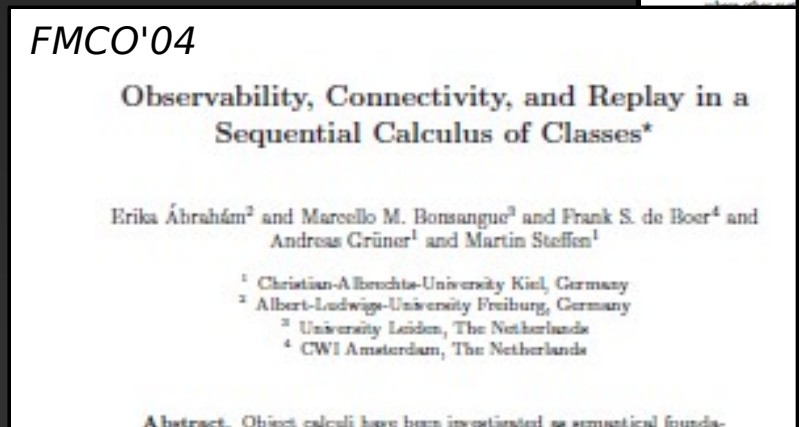
Alan Jeffrey*[1,2] and Julian Rathke[3]

[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



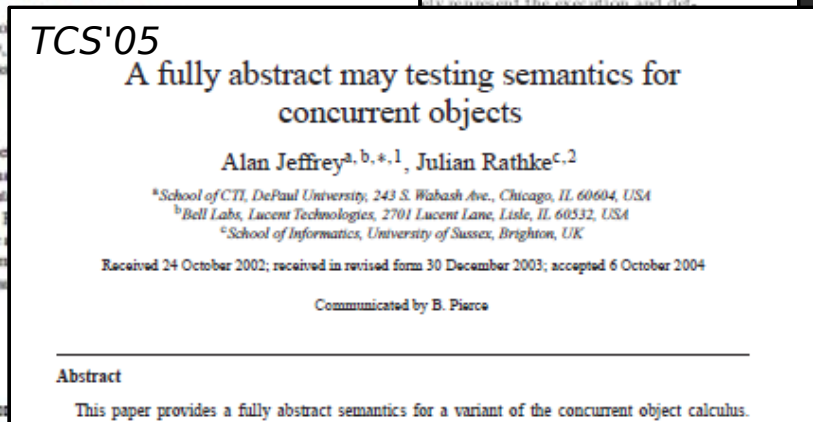
FMCO'04

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

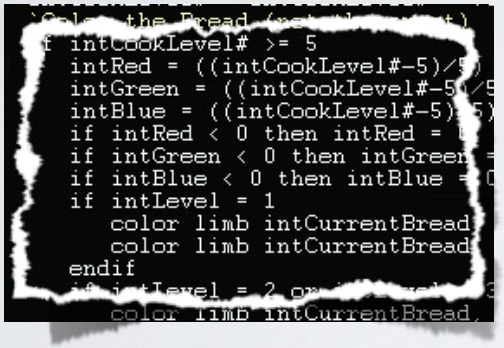
Received 24 October 2002; received in revised form 30 December 2003; accepted 6 October 2004

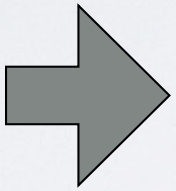
Communicated by B. Pierce

Abstract

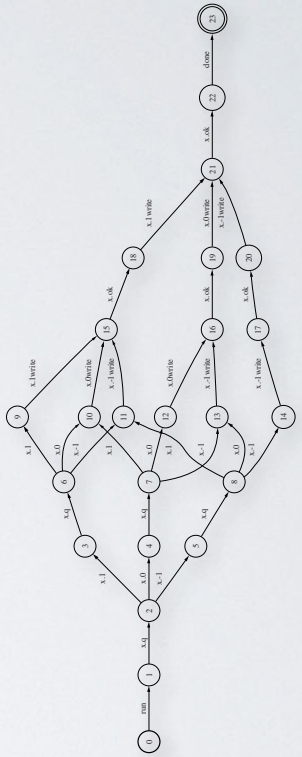
This paper provides a fully abstract semantics for a variant of the concurrent object calculus.

VERIFICATION





strategy



M_1, M_2 contextually equivalent $\iff [\![M_1]\!] = [\![M_2]\!] \iff \mathcal{A}_{M_1} \approx \mathcal{A}_{M_2}$

SOURCES OF UNDECIDABILITY

- arithmetic
- recursive definitions (datatypes and methods)
- storage of method-carrying objects in fields
- "higher-order" types

$$o_1 : \mathcal{I}_1, \quad \cdots \quad , o_k : \mathcal{I}_k \quad \vdash \quad M : \mathcal{I}$$

HIGHER-ORDER TYPES

bad

$$\begin{aligned} &\vdash \quad \bullet \to (\bullet \to \bullet) \\ &\vdash \quad ((\bullet \to \bullet) \to \bullet) \to \bullet \\ (\bullet \to \bullet) \to \bullet \quad &\vdash \end{aligned}$$

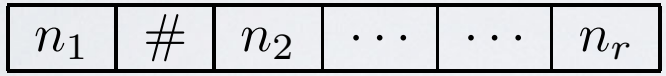
good

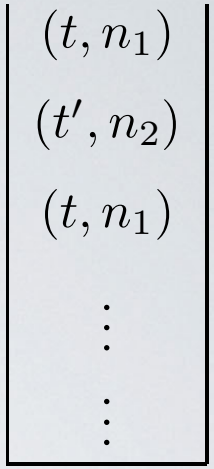
$$\mathsf{G} ::= \ \mathsf{void} \mid \mathsf{int} \mid \overrightarrow{f : \mathsf{G}}$$

$$\mathsf{L} ::= \ \mathsf{void} \mid \mathsf{int} \mid (\overrightarrow{\mathsf{f} : \mathsf{G}}, \ \overrightarrow{\mathsf{m} : \vec{\mathsf{G}} \to \mathsf{L}})$$

$$\mathsf{R} ::= \ \mathsf{void} \mid \mathsf{int} \mid (\overrightarrow{\mathsf{f} : \mathsf{G}}, \ \overrightarrow{\mathsf{m} : \vec{\mathsf{L}} \to \mathsf{G}})$$

AUTOMATA THEORY OVER INFINITE ALPHABETS





- RA

language equivalence (det)	*co-NP-complete*	[LICS'15]
bisimilarity	*PSPACE-complete*	[LICS'15]

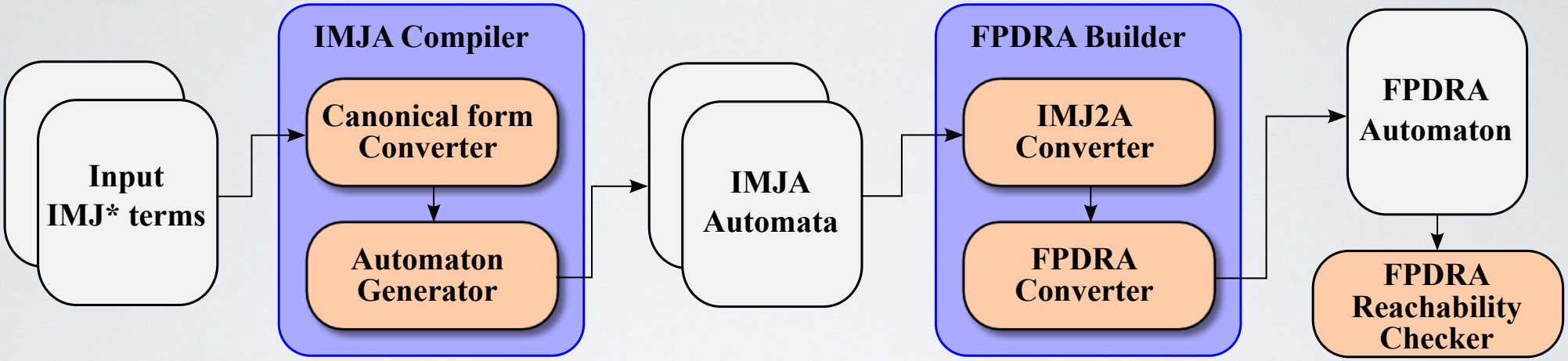
- PDRA

emptiness	*EXPTIME-complete*	[MFCS'14]
bisimilarity	*undecidable*	[LICS'15]
HO emptiness	*undecidable*	[MFCS'14]

- FPDRA

emptiness	*EXPTIME-complete*	[ICALP'12, MFCS'14]

CONEQCT (ATVA'15)



[POPL'14]
[ATVA'15]

[MFCS'14]
[ATVA'15]

[ICALP'12]
[MFCS'14]

Andrzej S. Murawski, Steven J. Ramsay, Nikos Tzevelekos: A Contextual Equivalence Checker for IMJ ∗. ATVA 2015: 234-240

COMPARISON

ADR09. A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *Proceedings of POPL*, pages 340–353. ACM, 2009.

BL05. N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *Proceedings of TLCA*, volume 3461 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2005.

DNB10. D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *Proceedings of ICFP*, pages 143–156. ACM, 2010.

KW06. V. Koutavas and M. Wand. Small bisimulations for reasoning about higher-order imperative programs. In *Proceedings of POPL*, pages 141–152. ACM, 2006.

PS98. A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In A. D. Gordon and A. M. Pitts, editors, *Higher-Order Operational Techniques in Semantics*, pages 227–273. Cambridge University Press, 1998.

WPH14. Yannick Welsch and Arnd Poetzsch-Heffter. A fully abstract trace-based semantics for reasoning about backward compatibility of class libraries. *Science of Computer Programming*, 92, Part B(0):129–161, 2014.

FUTURE WORK

- polymorphism
- soundness and incompleteness