Nominal games: a low-level semantics for open programs

Nikos Tzevelekos, Queen Mary Uni. of London

jointly with:
Andrzej Murawski & Steven Ramsay, Warwick
Dan Ghica, Birmingham
Thomas Cuvillier & Guilhem Jaber, QMUL

LOLA, Kyoto, July 2015

Supported by a Royal Academy of Engineering Research Fellowship
what this talk is about

The general setting is that of semantics of higher-order languages with effects: state, exceptions, polymorphism

We present recent work in nominal game semantics

1. games for nominal effects (ML)
2. games for Java programs (IMJ) and algorithmic games
3. operational games

→ games as a low-level semantics for full abstraction
what this talk is about

The general setting is that of semantics of higher-order languages with effects: state, exceptions, polymorphism

We present recent work in *nominal game semantics*

1. games for nominal effects (ML)
2. games for Java programs (IMJ) and algorithmic games
3. operational games

→ games as a low-level semantics for full abstraction
Setting: HO programs + nominal effects

\[ \begin{align*}
\Gamma \vdash () : \text{unit} & \quad \Gamma \vdash i : \text{int} & \quad \Gamma \vdash a : \text{ref } \theta & \quad \Gamma \vdash e : \text{exn} \\
\Gamma, x : \theta \vdash M : \theta' & \quad \Gamma \vdash M : \theta \rightarrow \theta' & \quad \Gamma \vdash N : \theta & \quad \Gamma \vdash MN : \theta' \\
\Gamma \vdash M, N : \text{ref } \theta & \quad \Gamma \vdash M = N : \text{int} & \quad \Gamma \vdash M, N : \text{exn} & \quad \Gamma \vdash M = N : \text{int} \\
\Gamma \vdash M : \theta & \quad \Gamma \vdash M : \text{ref } \theta & \quad \Gamma \vdash M : \text{ref } \theta & \quad \Gamma \vdash N : \theta \\
\Gamma \vdash \text{ref } M : \text{ref } \theta & \quad \Gamma \vdash !M : \theta & \quad \Gamma \vdash M := N : \text{unit} \\
\Gamma \vdash \text{exn}(\cdot) : \text{exn} & \quad \Gamma \vdash \text{raise } M : \theta & \quad \Gamma \vdash M : \theta, x : \text{exn} \vdash N : \theta \\
\Gamma \vdash M \text{ handle } x \Rightarrow N : \theta & \quad \Gamma \vdash M : \text{unit} \end{align*} \]
Operational semantics (examples)

\[ S, M \rightarrow S', M' \]

\[ S, (\lambda x. M)v \rightarrow S, M[v/x] \]

\[ S, a = a' \rightarrow S, 0/1 \quad a, a' \in \text{Loc}_\theta \]

\[ S, \text{ref } 0 \rightarrow S \cup [a \mapsto 0], a \quad a \in \text{Loc}_{\text{int}} \]

\[ S, \text{ref } (\lambda x. x+1) \rightarrow S \cup [a \mapsto \lambda x. x+1], a \quad a \in \text{Loc}_{\text{int} \rightarrow \text{int}} \]

let \( f = \text{ref } (\lambda x. x) \) in \( f := \lambda x. (!f)x; (!f)5 \)

\[ \rightarrow \ldots \rightarrow [a \mapsto \lambda x. (!a)x], (!a)5 \rightarrow [a \mapsto \lambda x. (!a)x], (\lambda x. (!a)x)5 \rightarrow \ldots \]
Models

How to assign denotations to programs,

\([-\] : Syntax \rightarrow \mathcal{M}\)

such that (ideally):

\[ M \cong M' \iff [M] = [M'] \]

same observable behaviour in every context
Nominal game semantics

- Computation is a 2-player game between:
  - **Opponent** (the environment), aka \( O \)
  - **Proponent** (the program), aka \( P \)
- Programs = *strategies* for \( P \)
- **Categories** of games
- models cast in *nominal sets*
- games played using *moves-with-stores*
- conditions for name *privacy and propagation*
Names and nominal sets

Nominal data: created fresh at will, compare for “="

- e.g. locations and exceptions
- can be modelled as names

Formally, this is done using nominal set theory:

- a universe built around atoms/names and atom-permutations
- all constructions involve a finite amount of atoms
- there is a canonical way to permute atoms inside constructs, take orbits, etc.

[ Gabbay & Pitts '02 ]
Arena games

\[ x_1 : \theta_1, \ldots, x_n : \theta_n \vdash M : \theta \]

- free variables
- input types
- program
- output type
Arena games

\[ x_1 : \theta_1, \ldots, x_n : \theta_n \vdash M : \theta \]

\[
[M] : [\theta_1, \ldots, \theta_n] \longrightarrow [\theta]
\]
Arenas of moves

\[
[M] : [\theta_1, \ldots, \theta_n] \rightarrow [\theta]
\]
Arenas of moves

\[ [M] : [\theta_1, \ldots, \theta_n] \rightarrow [\theta] \]

\[ [\text{unit}] = \{ * \} \]
\[ [\text{int}] = \{ 0, 1, -1, \ldots \} \]
\[ [\text{ref } \theta] = \{ a, b, \ldots \} \]

\[ a, b, \ldots \in \mathcal{N}_\theta \]
Arenas of moves

\[
[M] : [\theta_1, \ldots, \theta_n] \rightarrow [\theta]
\]

[unit] = \{ * \} = 1

[int] = \{ 0, 1, -1, \ldots \} = \mathbb{Z}

[ref θ] = \{ a, b, \ldots \} = \mathcal{A}_\theta

\mathcal{N}_\theta \text{ a set of names:}
- infinitely many
- comparable for equality only

\begin{align*}
a, b, \ldots & \in \mathcal{N}_\theta
\end{align*}
A higher-order arena

\[ \text{[int} \rightarrow \text{int]} \]

questions

\{ 0, 1, -1, \ldots \}

answers

\{ 0, 1, -1, \ldots \}
A higher-order arena

\[ [\text{int} \rightarrow \text{int}] = \mathbb{Z} \Rightarrow \mathbb{Z} \]

Questions: \{0, 1, -1, \ldots\}

Answers: \{0, 1, -1, \ldots\}
$\mathbb{Z} \Rightarrow \mathbb{Z} \quad \rightarrow \quad \mathbb{Z} \Rightarrow \mathbb{Z}$

e.g. $f : \text{int} \rightarrow \text{int} \vdash M : \text{int} \rightarrow \text{int}$
\[ \mathbb{Z} \Rightarrow \mathbb{Z} \quad \quad \quad \mathbb{Z} \Rightarrow \mathbb{Z} \]

e.g. \( f : \text{int} \rightarrow \text{int} \vdash M : \text{int} \rightarrow \text{int} \)

\( i, j, i', j' = 0, 1, -1, 2, -2, \ldots \)
Nominal games

⊢ ref 42 : ref int

1 → $A_{\text{int}}$

$a \in \mathcal{N}_{\text{int}}$
Nominal games

1 ⊢ ref 42 : ref int

1 → A_{int}

* → a^{(a,42)}

O,Q

P,A

a ∈ N_{int}
Nominal games

\[ \vdash \text{ref } 42 : \text{ref int} \]

\[ [\text{ref } 42] = \{ * \alpha^{(a,42)} \} \]

\[ a \in \mathcal{N}_{\text{int}} \]

\[ \begin{array}{c}
1 \\
\rightarrow \end{array} \quad \begin{array}{c}
\alpha^{(a,42)} \\
\rightarrow \end{array} \quad \begin{array}{c}
O,Q \\
P,A
\end{array} \]
Nominal games

\[ \vdash \text{ref } (\lambda x^{\text{int}}. x + 1) : \text{ref } (\text{int} \rightarrow \text{int}) \]
Nominal games

\[ \vdash \text{ref} (\lambda x^{\text{int}}. x + 1) : \text{ref} (\text{int} \rightarrow \text{int}) \]

1 \rightarrow \mathbb{A}_{\text{int} \rightarrow \text{int}}

\begin{align*}
* & \rightarrow a^{(a, \dagger)} \\
5 & \rightarrow a^{(a, \dagger)} \\
6 & \rightarrow a^{(a, \dagger)} \\
\end{align*}

\begin{align*}
O,Q & \rightarrow P,A \\
O,Q & \rightarrow P,A
\end{align*}

\( a \in \mathcal{N}_{\text{int} \rightarrow \text{int}} \)
$\vdash \text{ref } (\lambda x^\text{int}. x+1) : \text{ref } (\text{int} \rightarrow \text{int})$

Nominal games

1 $\longrightarrow$ $\text{A}_{\text{int} \rightarrow \text{int}}$

* $\leftarrow$

$\alpha (a, \dag)$

5 $(a, \dag)$

6 $(a, \dag)$

7 $(a, \dag)$

8 $(a, \dag)$

$O,Q$

$P,A$

$a \in \mathcal{N}_{\text{int} \rightarrow \text{int}}$
Nominal games

\[ \vdash \text{ref } (\lambda x^{\text{int}}. x + 1) : \text{ref } (\text{int}\rightarrow\text{int}) \]

\[ \left[ \text{ref } \lambda x. x + 1 \right] = \{ * \overset{\alpha^{(a,\dagger)}}{\rightarrow} 5^{(a,\dagger)} \overset{\alpha^{(a,\dagger)}}{\rightarrow} 6^{(a,\dagger)} \overset{\alpha^{(a,\dagger)}}{\rightarrow} 7^{(a,\dagger)} \overset{\alpha^{(a,\dagger)}}{\rightarrow} 8^{(a,\dagger)} \ldots \} \]
$\vdash \text{ref } (\lambda x^\text{int}.x+1) : \text{ref } (\text{int} \to \text{int})$

Nominal games

1 $\rightarrow$ A_{\text{int} \to \text{int}}

* $\rightarrow$ a_{(a, \dagger)}

5 $\rightarrow$ a_{(a, \dagger)}

6 $\rightarrow$ a_{(a, \dagger)}

O,Q

P,A

a \in \mathcal{N}_{\text{int} \to \text{int}}
⊢ ref (\(\lambda x^{\text{int}}. x + 1\)) : ref (\(\text{int} \to \text{int}\))

Nominal games

\(a \in \mathcal{N}_{\text{int} \to \text{int}}\)
Nominal games

\[ \vdash \text{ref } (\lambda x^{\text{int}}.x + 1) : \text{ref } (\text{int} \to \text{int}) \]

\[ \begin{array}{c}
1 \quad \text{A}_{\text{int} \to \text{int}} \\
* \quad a^{(a,\dagger)} \quad O,Q \\
P,A \quad \end{array} \]

\[ \left[ \text{ref } \lambda x.x + 1 \right] \supseteq \left\{ *, a^{(a,\dagger)}, 5^{(a,\dagger)}, 6^{(a,\dagger)}, 7^{(a,\dagger)}, 7^{(a,\dagger)}, \ldots \right\} \]

\[ \begin{array}{c}
OQ \quad PA \\
OQ \quad PA \\
OQ \quad PQ \\
7^{(a,\dagger)} \quad P,Q \\
\end{array} \]
Quiz

\[ \begin{array}{c}
\text{??}
\end{array} \] \supseteq \{ *_{(a,\dagger)}, 5_{(a,\dagger)}, 6_{(a,\dagger)}, 7_{(a,\dagger)}, 8_{(a,\dagger)}, \ldots \} 

\begin{array}{cccc}
OQ & PA & OQ & PA & OQ & PQ \\
\end{array}
Quiz

⊢ let \( l = \text{ref} \lambda x^{\text{int}}.x \) in

\((l := \lambda x^{\text{int}}.(l := \lambda y^{\text{int}}.y+1); \ x+1); \ l : \text{ref}(\text{int} \rightarrow \text{int})\)
Model for nominal references

Games with names and moves-with-stores:

\[ S = \{ (a,4), (b,c), (c,3), (d,\dagger), \ldots \} \]

- pointers to moves & stores
- name privacy made explicit

[H0 values!]

[ Laird '08, Murawski & T. '09, '11, '12 ]
Model for nominal references

Games with names and moves-with-stores:

\[ S = \{ (a,4), (b,c), (c,3), (d,\dagger), \ldots \} \]

- pointers to moves & stores
- name privacy made explicit

Composition requires nominal conditions:

- **privacy**: one strategy cannot guess the other's names
- **store access**: strategies can only access the parts of the store they know

[ Laird '08, Murawski & T. '09, '11, '12 ]
Model for nominal references

Games with names and moves-with-stores:

\[ S = \{ (a,4), (b,c), (c,3), (d,\dagger), \ldots \} \]

- pointers to moves & stores
- name privacy made explicit

Full abstraction:

\[ M \cong M' \iff [M] = [M'] \]

(in fact, only “complete” plays count)

[ Laird '08, Murawski & T. '09, '11, '12 ]
More nominal effects: exceptions

typed locations (location names)

\[
\begin{align*}
\Gamma \vdash () : \text{unit} & \quad \Gamma \vdash i : \text{int} & \quad \Gamma \vdash a : \text{ref } \theta \\
\Gamma, x : \theta \vdash M : \theta' & \quad \Gamma \vdash M : \theta \rightarrow \theta' & \quad \Gamma \vdash N : \theta \\
\Gamma \vdash \lambda x^A. M : \theta \rightarrow \theta' & \quad \Gamma \vdash M N : \theta' \quad \Gamma \vdash M N : \theta' \\
\Gamma \vdash M, N : \text{ref } \theta & \quad \Gamma \vdash M = N : \text{int} \quad \Gamma \vdash M = N : \text{int} \\
\Gamma \vdash M : \theta & \quad \Gamma \vdash M : \text{ref } \theta & \quad \Gamma \vdash M : \text{ref } \theta \\
\Gamma \vdash \text{ref } M : \text{ref } \theta & \quad \Gamma \vdash \text{!} M : \theta & \quad \Gamma \vdash N : \theta \\
\Gamma \vdash \text{exn }() : \text{exn} & \quad \Gamma \vdash \text{raise } M : \theta & \quad \Gamma, x : \text{exn } \vdash N : \theta \\
\Gamma \vdash M : \theta & \quad \Gamma \vdash M : \text{unit} & \quad \Gamma \vdash M \text{ handle } x \Rightarrow N : \theta
\end{align*}
\]
More nominal effects: exceptions

\[ \Gamma \vdash () : \text{unit} \quad \Gamma \vdash i : \text{int} \]
\[ \Gamma \vdash a : \text{ref } \varnothing \quad \Gamma \vdash e : \text{exn} \]

\[ \Gamma, x:\varnothing \vdash M : \varnothing' \]
\[ \Gamma \vdash \lambda x^\varnothing. M : \varnothing \rightarrow \varnothing' \]
\[ \Gamma \vdash M : \varnothing \rightarrow \varnothing' \quad \Gamma \vdash N : \varnothing \]
\[ \Gamma \vdash MN : \varnothing' \]

\[ \Gamma \vdash M, N : \text{ref } \varnothing \]
\[ \Gamma \vdash M = N : \text{int} \]

\[ \Gamma \vdash M : \varnothing \]
\[ \Gamma \vdash \text{ref } M : \text{ref } \varnothing \]
\[ \Gamma \vdash \text{ref } M : \text{ref } \varnothing \]
\[ \Gamma \vdash !M : \varnothing \]
\[ \Gamma \vdash !M : \varnothing \]
\[ \Gamma \vdash M := N : \text{unit} \]

\[ \Gamma \vdash M : \varnothing \]
\[ \Gamma \vdash \text{exn}() : \text{exn} \]
\[ \Gamma \vdash \text{raise } M : \varnothing \]
\[ \Gamma \vdash \text{exn}() : \text{exn} \]
\[ \Gamma \vdash M : \varnothing \]
\[ \Gamma \vdash M : \varnothing \]

\[ \Gamma, x: \text{exn} \vdash N : \varnothing \]
\[ \Gamma \vdash M \text{ handle } x \Rightarrow N : \varnothing \]
Operational semantics (examples)

\[ S, M \rightarrow S', M' \]

\[ S, (\lambda x. M) v \rightarrow S, M[v/x] \]

\[ S, a = a' \rightarrow S, 0/1 \quad S, e = e' \rightarrow S, 0/1 \]

\[ S, \text{exn()} \rightarrow S \uplus \{e\}, e \]

\[ S, (\lambda x. M)(\text{raise } e) \rightarrow S, \text{raise } e \]

\[ S, v \text{ handle } x \Rightarrow M \rightarrow S, v \]

\[ S, (\text{raise } e) \text{ handle } x \Rightarrow M \rightarrow S, M[e/x] \]

\( S \) stores locations + their values, & a set of exceptions \( e, e' \in \text{Exn} \)
let $x = \text{exn}()$ in $\lambda z. \text{raise } x$  \neq_{\text{unit} \rightarrow \text{unit}} \lambda z. \text{raise } (\text{exn}())$

\[
\begin{align*}
S, (\lambda x. M) v & \rightarrow S, M[v/x] \\
S, a = a' & \rightarrow S, 0/1 \\
S, \text{exn()} & \rightarrow S \uplus \{e\}, e \\
S, (\lambda x. M)(\text{raise } e) & \rightarrow S, \text{raise } e \\
S, v \text{ handle } x \Rightarrow M & \rightarrow S, v \\
S, (\text{raise } e) \text{ handle } x \Rightarrow M & \rightarrow S, M[e/x]
\end{align*}
\]
Model for exceptions

Cater for *exceptional answer* moves: $e!$

- can answer any open question
- model otherwise unchanged

[Murawski & T. '14]
Model for exceptions

Cater for exceptional answer moves: \( e! \)

- can answer any open question
- model otherwise unchanged

Private exceptions:

\[
\begin{align*}
\text{let } x &= \text{exn}() \text{ in } \lambda z. \text{raise } x & \equiv & \lambda z. \text{raise } (\text{exn}())
\end{align*}
\]

\( \rightarrow \) restrict by fresh-exception propagation

model fully abstract in both cases

[ Murawski & T. '14 ]
Names for polymorphism

Polymorphic values modelled by names:

• no computational content
• polymorphic values naturally permutable

[ Laird '10; Cuvillier, Jaber & T. ]
Names for polymorphism

Polymorphic values modelled by names:
- no computational content
- polymorphic values naturally permutable

\[
\begin{align*}
\text{[int} \rightarrow \text{int}] &= \alpha \\
\forall X. X \rightarrow X] &= \alpha
\end{align*}
\]

Names for:
- type variables $\alpha \in \mathcal{N}_{\text{TVar}}$
- polymorphic values $p \in \mathcal{N}_{\text{poly}}$

[ Laird '10; Cuvillier, Jaber & T. ]
what this talk is about

The general setting is that of semantics of higher-order languages with effects: state, exceptions, polymorphism

We explore recent work in nominal game semantics

1. games for nominal effects (ML)
2. **games for Java programs (IMJ)** and algorithmic games
3. operational games

→ games as a low-level semantics for full abstraction
Games for Java programs

We can view objects as HO-references

- use names, moves-with-store, move/store pointers
- not really HO programs: no functions, only methods
  - we can remove pointers to moves
- methods cannot be updated
  - we can remove pointers altogether

[ Murawski & T. '14 ]
Interface Middleweight Java (IMJ)

Types \( \theta ::= \text{void} | \text{int} | \mathcal{I} \)

Interface definitions

\( \Theta ::= \emptyset | (f : \theta), \Theta | (m : \overline{\theta} \rightarrow \theta), \Theta \)

Interface tables

\( \Delta ::= \emptyset | (\mathcal{I} : \Theta), \Delta | (\mathcal{I} \langle \mathcal{I} \rangle : \Theta), \Delta \)

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

\( \Theta ::= \emptyset \mid (f : \theta), \Theta \mid (m : \overline{\theta} \rightarrow \theta), \Theta \)

Interface tables

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

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

- Objects, inheritance, casting, interfaces
Interface Middleweight Java (IMJ)

Types

\[ \theta ::= \text{void} \mid \text{int} \mid I \]

Interface definitions

\[ \Theta ::= \emptyset \mid (f : \theta), \Theta \mid (m : \overline{\theta} \rightarrow \theta), \Theta \]

Interface tables

\[ \Delta ::= \emptyset \mid (I : \Theta), \Delta \mid (I\langle I \rangle : \Theta), \Delta \]

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

- Objects, inheritance, casting, interfaces
Interface Middleweight Java (IMJ)

Terms

\[ M :::= \ \text{skip} \mid n \mid \text{null} \mid x \mid i \mid M \oplus M \mid \text{if } M \quad M \quad M \]

\[ \mid \text{let } x = M \text{ in } M \mid M = M \mid (\mathcal{I})M \mid \text{while } M \quad M \]

\[ \mid \text{new}(x : \mathcal{I} ; M) \mid M.f \mid M.f := M \mid M.m(\overline{M}) \]

Method implementations

\[ M :::= \ \emptyset \mid (m : \lambda x. M), M \]

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

- Objects, inheritance, casting, interfaces
**IMJ: operational semantics**

<table>
<thead>
<tr>
<th>$S, M \rightarrow S', M'$</th>
<th>$S$ stores object (names) + their types and values</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S, \text{let } x = v \text{ in } M \rightarrow S, M[v/x]$</td>
<td>Obj : set of objects</td>
</tr>
<tr>
<td>$S, a = a' \rightarrow S, 0/1$</td>
<td>$a, a' \in \text{Obj}$</td>
</tr>
<tr>
<td>$S, (I)a \rightarrow S, a$ if $S(a) = I' \leq I$</td>
<td></td>
</tr>
<tr>
<td>$S, \text{new}(x:I; M) \rightarrow S \uplus {(a, I, (V_I, M[a/x]))}, a$</td>
<td>$V_I$: default field values</td>
</tr>
</tbody>
</table>
Examples

\[ x : \text{Var} \vdash x.\text{val} + 1 : \text{int} \]

\[ \text{Var} : \{ \text{val} : \text{int} \} \]

\[ O : \alpha^{(a.\text{val} = 5)} \]

\[ P : 6^{(a.\text{val} = 5)} \]
Examples

\[ x : \text{Var} \vdash x.\text{val} + 1 : \text{int} \]

\[ \text{Var} : \{ \text{val} : \text{int} \} \]

\[ \begin{array}{ll}
\text{O} : \alpha^{(a.\text{val} = 5)} & \quad \text{O} : \alpha^{(a.\text{val} = 8)} \\
\text{P} : 6^{(a.\text{val} = 5)} & \quad \text{P} : 9^{(a.\text{val} = 8)}
\end{array} \]
Examples

\[ x : \text{Var} \vdash x.\text{val} + 1 : \text{int} \]

\[ \text{Var} : \{ \text{val} : \text{int} \} \]

\[ O : a^{(a.\text{val} = 5)} \quad P : 6^{(a.\text{val} = 5)} \]

\[ O : a^{(a.\text{val} = 8)} \quad P : 9^{(a.\text{val} = 8)} \]

\[ O : a^{(a.\text{val} = 3)} \quad P : 4^{(a.\text{val} = 3)} \]

\[
\left[ x : \text{Var} \vdash x.\text{val} + 1 : \text{int} \right] = \{ a^{(a.\text{val} = i)} (i+1)^{(a.\text{val} = i)} \} \]
Examples

\[ x : \text{Var}, f : \text{Fun} \vdash f.\text{val} (x.\text{val}) + 1 : \text{int} \]

\text{Var} : \{ \text{val} : \text{int} \} \quad \text{Fun} : \{ \text{val} : \text{int} \rightarrow \text{int} \}
Examples

\[ x : \text{Var}, \, f : \text{Fun} \vdash f.\text{val} (x.\text{val}) + 1 : \text{int} \]

\[ \text{Var} : \{ \text{val} : \text{int} \} \quad \text{Fun} : \{ \text{val} : \text{int} \rightarrow \text{int} \} \]

\[ O : (a, f) \quad (a.\text{val} = 4) \]
Examples

\[
x : \text{Var}, \quad f : \text{Fun} \quad \vdash \quad f.\text{val} (x.\text{val}) + 1 : \text{int}
\]

\[
\text{Var} : \{ \text{val} : \text{int} \} \quad \text{Fun} : \{ \text{val} : \text{int} \rightarrow \text{int} \}
\]

\[
\begin{align*}
\text{O} : \quad & (a,f) (a.\text{val} = 4) \\
\text{P} : \quad & \text{call } f.\text{val}(4) (a.\text{val} = 4)
\end{align*}
\]
Examples

\[ x : \text{Var}, f : \text{Fun} \vdash f.\text{val}(x.\text{val}) + 1 : \text{int} \]

\[
\begin{align*}
\text{Var} & : \{ \text{val} : \text{int} \} \\
\text{Fun} & : \{ \text{val} : \text{int} \rightarrow \text{int} \}
\end{align*}
\]

\[
\begin{align*}
O & : (a,f)^{(a.\text{val} = 4)} \\
P & : \text{call } f.\text{val}(4)^{(a.\text{val} = 4)} \\
O & : \text{ret } f.\text{val}(50)^{(a.\text{val} = 73)}
\end{align*}
\]
Examples

\[ x : \text{Var}, f : \text{Fun} \vdash f.\text{val}(x.\text{val}) + 1 : \text{int} \]

\[ \text{Var} : \{ \text{val} : \text{int} \} \quad \text{Fun} : \{ \text{val} : \text{int} \rightarrow \text{int} \} \]

\[ \text{O} : (a,f)^{(a.\text{val} = 4)} \]

\[ \text{P} : \text{call } f.\text{val}(4)^{(a.\text{val} = 4)} \]

\[ \text{O} : \text{ret } f.\text{val}(50)^{(a.\text{val} = 73)} \]

\[ \text{P} : 51^{(a.\text{val} = 73)} \]
Examples

\[ x : \text{Var}, \ f : \text{Fun} \vdash f.\text{val} (x.\text{val}) + 1 : \text{int} \]
\[ \text{Var} : \{ \text{val} : \text{int} \} \quad \text{Fun} : \{ \text{val} : \text{int} \rightarrow \text{int} \} \]

\[ O : (a,f)^{(a.\text{val} = 4)} \]
\[ P : \text{call } f.\text{val}(4)^{(a.\text{val} = 4)} \]
\[ O : \text{ret } f.\text{val}(50)^{(a.\text{val} = 73)} \]
\[ P : 51^{(a.\text{val} = 73)} \]

\[
\begin{align*}
x : \text{Var}, \ f : \text{Fun} \vdash f.\text{val} (x.\text{val}) + 1 : \text{int} \\
= \{ (a,f)^{(a.\text{val} = i)} \text{ call } f.\text{val}(i)^{(a.\text{val} = i)} \text{ ret } f.\text{val}(j)^{(a.\text{val} = i') \text{ (j+1)}^{(a.\text{val} = i')}} \}
\end{align*}
\]
**IMJ example: game semantics**

\[ M_1 : \text{let } u = \text{new}(\text{Var}_{\text{Emp}}) \text{ in } \]
\[ \text{new}(M_1) : \text{Cell} \]

\[ M_1 : \text{get: } \lambda(). u.\text{val}, \]
\[ \text{set: } \lambda y. u.\text{val} := y \]

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

\[
\begin{align*}
[M_1] &= * a^{\Sigma_0} (\text{call } a.\text{get}(\Sigma_0) \text{ ret } a.\text{get}(\text{nul})^{\Sigma_0})^* \\
&\quad \text{call } a.\text{set}(a_1)^{\Sigma_1} \text{ ret } a.\text{set}()^{\Sigma_1} \\
&\quad (\text{call } a.\text{get}(\Sigma_1) \text{ ret } a.\text{get}(a_1)^{\Sigma_1})^* \\
&\quad \text{call } a.\text{set}(a_2)^{\Sigma_2} \text{ ret } a.\text{set}()^{\Sigma_2} \ldots
\end{align*}
\]

\[ \Sigma_i = \{ a \mapsto (\text{Cell, } \_ ) \} \cup \{ a_j \mapsto (\text{Empty, } \_), 1 \leq j \leq i \} \]
**IMJ example: game semantics**

\[
\begin{align*}
M_2 &: \quad \text{let } b = \text{new}(\text{Var}_{\text{Int}}) \text{ in} \\
& \quad \text{let } u_1 = \text{new}(\text{Var}_{\text{Emp}}) \text{ in} \\
& \quad \text{let } u_2 = \text{new}(\text{Var}_{\text{Emp}}) \text{ in} \\
& \quad \text{new}(M_2) : \text{Cell} \\
M_2 &: \quad \text{get: } \lambda(). \text{ if } b.\text{val} \\
& \quad \text{then } b.\text{val} := 0; u_1.\text{val} \\
& \quad \text{else } b.\text{val} := 1; u_2.\text{val}, \text{ set: } \lambda y. u_1.\text{val} := y; \text{ } u_2.\text{val} := y
\end{align*}
\]

\[
\begin{align*}
\Sigma_1 &= \{ a \mapsto \text{(Cell, _)} \} \cup \{ a_j \mapsto \text{(Empty, _)}, 1 \leq j \leq i \} \\
\end{align*}
\]

\[
\begin{align*}
[M_1] &= * a^{\Sigma_0} \text{ ( call } a.\text{get()}^{\Sigma_0} \text{ ret } a.\text{get(nul)}^{\Sigma_0} )* \\
& \quad \text{call } a.\text{set}(a_1)^{\Sigma_1} \text{ ret } a.\text{set()}^{\Sigma_1} \\
& \quad \text{( call } a.\text{get()}^{\Sigma_1} \text{ ret } a.\text{get}(a_1)^{\Sigma_1} )* \\
& \quad \text{call } a.\text{set}(a_2)^{\Sigma_2} \text{ ret } a.\text{set()}^{\Sigma_2} \ldots = [M_2]
\end{align*}
\]

* Koutavas & Wand' 07
Games and equivalence algorithmically

\[ M \] Programs

\[ \llbracket M \rrbracket \] Game Semantics

\[ A(M) \] Automata
Games and equivalence algorithmically

\[ M \cong M' \iff \llbracket M \rrbracket = \llbracket M' \rrbracket \iff A \sim A' \]
Games and equivalence algorithmically

The passage can be done automatically → decision procedure for program equivalence

- Restrict to the finitary fragment (bounded datatypes)
- Full classification based on types (at each type, either the problem is undecidable or we get a procedure)
- employ automata are over infinite alphabets

\[ M \simeq M' \iff [M] = [M'] \iff A \sim A' \iff A \otimes A' = 0 \]

[ Murawski & T. '12; Murawski, Ramsay & T.' 15 ]
Decidable types (IMJ)

\[ x : L \vdash M : R \]

\[
G ::= \text{void} \mid \text{int} \mid (f : G)
\]

\[
L ::= \text{void} \mid \text{int} \mid (f : G, m : G \rightarrow L)
\]

\[
R ::= \text{void} \mid \text{int} \mid (f : G, m : L \rightarrow G)
\]
Conect: a contextual equivalence checking tool for Interface Middleweight Java

Requirements
The checker runs on the .NET platform (>4.5), and hence requires a recent implementation of the .NET Common Language Infrastructure (CLI) to be installed on your system.

- On Linux or Mac we recommend Xamarin’s "Mono": http://www.mono-project.com/download/

Installation
1. Download the latest assemblies from downloads. All the required assemblies are packaged together in a zip file named "conect-xxx.zip" where "XXX" denotes the revision number.
2. Unzip to any convenient location, this creates a new directory "conect-XXX" in which resides the executable "conect.exe".
3. To verify that all is well, on the command line navigate to the directory "conect-XXX" and run the command:
   - "conect.exe" on Windows or,
   - "mono conect.exe" on Linux or Mac. If the installation is working correctly, the usage message will be printed out to the terminal.

Usage
On Windows, to check the equivalence of two IMJ terms defined in the file "terms.inp", run:

```
> ./conect.exe /path/to/terms.inp
```

On Linux or Mac you should prefix this command by "mono" (and use appropriate slashes):

```
> mono ./conect.exe /path/to/terms.inp
```

A number of example inputs are bundled with the installation. For example, after navigating to the root of the directory "conect-XXX", to verify the "extended types" equivalence adapted from Benton and Leperchey’s "Relational Reasoning in a Nominal Semantics for Storage" on Mac, run:

```
> mono ./conect.exe inputs/imp2.imj
```

See Syntax for a detailed description of the syntax of the input file format.

The tool can be configured using some command-line options:

- `-maxint <int>`: set the value of maxint to <int>
\( M_1 \equiv \text{let } v = \text{new } \{ _\_ : \text{Var}_{\text{Empty}} \} \text{ in} \\
\text{new } \{ _\_ : \text{Cell}; \\
\text{get} : \lambda _\_. v.\text{val}, \\
\text{set} : \lambda y. \text{if } y=\text{null} \text{ then div else } v.\text{val}:=y \} \)

\( M_2 \equiv \text{let } b = \text{new } \{ _\_ : \text{Var}_{\text{Int}} \} \text{ in} \\
\text{let } v = \text{new } \{ _\_ : \text{Var}_{\text{Empty}} \} \text{ in} \\
\text{let } w = \text{new } \{ _\_ : \text{Var}_{\text{Empty}} \} \text{ in} \\
\text{new } \{ _\_ : \text{Cell}; \\
\text{get} : \lambda _\_. \text{if } b.\text{val}=1 \text{ then } (b.\text{val}:=0; v.\text{val}) \\
\text{else } (b.\text{val}:=1; w.\text{val}), \\
\text{set} : \lambda y. \text{if } y=\text{null} \text{ then div} \\
\text{else } v.\text{val}:=y; w.\text{val}:=y \} \)
\[ M_1 \equiv \text{let } v = \text{new } \{ \_ : \text{VarEmpty} \} \text{ in} \\
\text{new } \{ \_ : \text{Cell} ; \\
\text{get } : \lambda_. \text{ v.val} . \\
\text{set } : \lambda y. \text{ if } y = \text{null} \text{ then div else } v.\text{val}:=y \} \]

\[ M_2 \equiv \text{let } b = \text{new } \{ \_ : \text{VarInt} \} \text{ in} \\
\text{let } v = \text{new } \{ \_ : \text{VarEmpty} \} \text{ in} \\
\text{let } w = \text{new } \{ \_ : \text{VarEmpty} \} \text{ in} \\
\text{new } \{ \_ : \text{Cell} ; \\
\text{get } : \lambda_. \text{ if } b.\text{val}=1 \text{ then } (b.\text{val}:=0; \text{ v.val}) \text{ else } (b.\text{val}:=1; \text{ w.val}) , \\
\text{set } : \lambda y. \text{ if } y = \text{null} \text{ then div else } v.\text{val}:=y; \text{ w.val}:=y \} \]
what this talk is about

The general setting is that of semantics of higher-order languages with effects: state, exceptions, polymorphism

We explore recent work in nominal game semantics

1. games for nominal effects (ML)
2. games for Java programs (IMJ) and algorithmic games
3. operational games

→ games as a low-level semantics for full abstraction
Games and traces

operational/denotational models

from arenas and plays to traces of “open” transition systems
Games and traces

operational/denotational models

from arenas and plays to traces of “open” transition systems

---

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

Alan Jeffrey*1,2 and Julian Rathke3

1 Bell Labs, Lucent Technologies, Chicago, IL, USA
2 DePaul University, Chicago, IL, USA
3 University of Sussex, Brighton, UK

Abstract. We introduce an expressive yet semantically clean core Java-like language, Java Jr., and provide it with a formal operational semantics based on traces of observable actions which represent interaction across package boundaries. A detailed example based on the Observer Pattern is used to demonstrate the intuitive character of the semantic model. We also show that our semantic trace equivalence is fully-abstract with respect to a natural notion of testing equivalence for object systems. This is the first such result for a full class-based OO-language with inheritance.

1 Introduction

Operational semantics as a modelling tool for program behaviour originated in the early 1960s in early work of McCarthy [19] and found some popularity in modelling programming languages such as ALGOL and LISP [20, 30, 13] and the lambda-calculus, [18]. Later, this approach to modelling was championed by Plotkin [25, 26] and has since been applied extensively and successfully for providing semantic descriptions of simple programming languages and computational models [31, 21, 22, 15, 29, 1, 11, 8]. As these modelling techniques began to be applied to larger scale languages, their semantic descriptions became more complex [27, 28, 4, 9, 6, 3].

There has been a considerable research effort towards formalising operational behaviour of Java and Java-like languages, for example [4, 11, 16, 6, 9, 24, 3]. Indeed [3] is
Games and traces

operational/denotational models

from arenas and plays to traces of “open” transition systems

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

Alan Jeffrey*1,2 and Julian Rathke3

1 Bell Labs, Lucent Technologies, Chicago, IL, USA
2 DePaul University, Chicago, IL, USA
3 University of Sussex, Brighton, UK

Papers from the Full Abstraction Factory


Games and traces

operational/denotational models

from arenas and plays to traces of “open” transition systems

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

J. Laird∗
e-mail:jiml@sussex.ac.uk
Dept. of Informatics, University of Sussex, UK.

Abstract. We describe a fully abstract trace semantics for a functional language with locally declared general references (a fragment of Standard ML). It is based on a bipartite LTS in which states alternate between program and environment configurations and labels carry only (sets of) basic values, location and pointer names. Interaction between programs and environments is either direct (initiating or terminating subprocedures) or indirect (by the overwriting of shared locations): actions reflect this by carrying updates to the shared part of the store.

The trace-sets of programs and contexts may be viewed as deterministic strategies and counter-strategies in the sense of game semantics: we prove soundness of the semantics by showing that the evaluation of a program in an environment tracks the interaction between the corresponding strategies. We establish full abstraction by proving a definability result: every bounded deterministic strategy of a given type is the trace-set of a configuration of that type.

1 Introduction

The conjunction of functional programming and general references is a powerful one — for example, it can describe both object-oriented and aspect-oriented [11] computation by translation. So it is not, perhaps, surprising that the behaviour of functional programs with locally bound references is difficult to reason about; they may exhibit a variety of subtle phenomena such as aliasing, and self-referencing and self-updating variables. In some respects, the higher-order “pointer-passing” exhibited by such programs is analogous to process-passing in a concurrent setting. The most significant differences between pointer-
Games and traces

operational/denotational models

from arenas and plays to traces of “open” transition systems
Games and traces

operational/denotational models
from arenas and plays to traces of “open” transition systems
Games and traces

operational/denotational models

from arenas and plays to traces of “open” transition systems
Games vs traces

Games are denotational:
- categories of arenas & strategies
- no access to local “small” steps

Traces are operational:
- defined directly via the operational semantics
- compositionality is a “sanity check”
Operational “flat” games

A marriage of games and traces whereby:

• pointers are replaced by names:
  
  plays are (bracketed) sequences of name-calls and -returns

• the arena structure is removed altogether:

  arenas become flat: integers, units and names (functions, references, exceptions, etc.)

  composition becomes more involved, though, as move/name ownership is now dynamic
$\mathbb{Z} \Rightarrow \mathbb{Z} \quad \rightarrow \quad \mathbb{Z} \Rightarrow \mathbb{Z}$

$O$ question

$P$ questions

$O$ answers

$0, 1, -1, \ldots$

$0, 1, -1, 2, \ldots$

$P$ answer

$O$ questions

$P$ answers

$0, 1, -1, \ldots$

$0, 1, -1, 2, \ldots$
$\mathbb{Z} \Rightarrow \mathbb{Z} \quad \rightarrow \quad \mathbb{Z} \Rightarrow \mathbb{Z}$

- $P$ questions
- $O$ questions
- $P$ answers
- $O$ answers

Diagram showing the relationships between $0, 1, -1, 2, \ldots$ and $P$ questions leading to $O$ answers, and $O$ questions leading to $P$ answers.
\[ \mathbb{Z} \Rightarrow \mathbb{Z} \quad \rightarrow \quad \mathbb{Z} \Rightarrow \mathbb{Z} \]

O question \( \rightarrow \) f \( \text{Red Triangle} \)

P answer \( \rightarrow \) g \( \text{Blue Triangle} \)
\( \mathbb{Z} \Rightarrow \mathbb{Z} \rightarrow \mathbb{Z} \Rightarrow \mathbb{Z} \)

\[ f \text{ call}(f,8) \text{ ret}(f,i) \quad g \text{ call}(g,j) \text{ ret}(g,i) \]

\[ (\text{let } x = f8 \text{ in } \lambda z. x) \]
Nominal games

\[ \vdash \mathit{ref} \ (\lambda x^\mathbb{int}. x+1) : \mathit{ref} \ (\mathbb{int} \to \mathbb{int}) \]

\[ [\mathit{ref} \ \lambda x. x+1] \supseteq \{ \ast \ a^{(a, \dagger)} \ 5^{(a, \dagger)} \ 6^{(a, \dagger)} \ 7^{(a, \dagger)} \ 8^{(a, \dagger)} \ \ldots \} \]

\* \ a^{(a,f)} \ \text{call} \ (f, 5)^{(a,f')} \ \text{ret} \ (f, 6)^{(a,f'')} \ \text{call} \ (f, 7)^{(a,f''')} \ \text{ret} \ (f, 8)^{(a,f''''})
Nominal games

\[ \vdash \text{ref } (\lambda x^{\text{int}}. x+1) : \text{ref } (\text{int}\rightarrow\text{int}) \]

\[
\left[ \text{ref } \lambda x. x+1 \right] \supseteq \{ \ast \alpha^{(a,\dagger)} 5^{(a,\dagger)} 6^{(a,\dagger)} 7^{(a,\dagger)} 7^{(a,\dagger)} \ldots \} \]

\[
\ast \alpha^{(a,f)} \text{ call } (f,5)^{(a,f')} \text{ ret } (f,6)^{(a,f'')} \text{ call } (f'',7)^{(a,f''')} \text{ call } (f',7)^{(a,f''''')}\]

Nom. games for programs: what's next

Concurrency, polymorphism + references

Model checking beyond equivalence (logics?)

Games & traces

- “simple” games for non-experts (!)
- transfer of trace reasoning techniques
- lift of FO techniques to HO
Nom. games for programs: what's next

Concurrency, polymorphism + references

Model checking beyond equivalence (logics?)

Games & traces

- “simple” games for non-experts (!)
- transfer of trace reasoning techniques
- lift of FO techniques to HO

Thanks!