the Nominal Games Company

presents

Andrzej Murawski **Steven Ramsay** and **Nikos Tzevelekos**

in

Game Semantic Analysis of Equivalence in Interface Middleweight Java

an epic story

MFPS, Nijmegen, June 2015

what this talk is about

We apply game semantics in program verification

→ characterising equivalence of Java programs

We work in an open fragment of MJ by use of interfaces

- open programs → need for games
- Java programs → simpler games

Full type-based characterisation:

- queue machine encodings (negative cases)
- pushdown (register) automata (algorithms)

Interface Middleweight Java (IMJ)

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

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

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

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

Interface tables

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

interface ident.

field identifier

method identif.

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

- Objects, inheritance, casting, interfaces

Interface Middleweight Java (IMJ)

Terms

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

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

Interface Middleweight Java (IMJ)

Terms

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

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

$$\frac{\Delta|\Gamma \vdash M : \text{int} \quad \Delta|\Gamma \vdash M', M'' : \theta}{\Delta|\Gamma \vdash \text{if } M \text{ then } M' \text{ else } M'' : \theta} \qquad \frac{\bigwedge_{i=1}^{n}(\Delta|\Gamma \uplus \{\vec{x}_i : \vec{\theta}_i\} \vdash M_i : \theta_i)}{\Delta|\Gamma \vdash \mathcal{M} : \{\mathsf{m}_i : \vec{\theta}_i \to \theta_i \mid 1 \le i \le n\}}$$

$$\frac{\Delta|\Gamma \vdash M, M' : I}{\Delta|\Gamma \vdash M = M' : \text{int}} \qquad \frac{\Delta|\Gamma \vdash M : \text{void} \quad \Delta|\Gamma \vdash M' : \theta}{\Delta|\Gamma \vdash M; M' : \theta} \qquad \frac{\Delta|\Gamma \vdash M : I \quad \Delta|\Gamma \vdash M' : \theta}{\Delta|\Gamma \vdash M.\mathsf{f} := M' : \text{void}}\ {\scriptstyle \Delta(I).\mathsf{f}=\theta}$$

$$\frac{\Delta|\Gamma \vdash M : I}{\Delta|\Gamma \vdash M.\mathsf{f} : \theta}\ {\scriptstyle \Delta(I).\mathsf{f}=\theta} \qquad \frac{\Delta|\Gamma \vdash M : I'}{\Delta|\Gamma \vdash (I)M : I}\ {\scriptstyle \Delta \vdash I \le I' \vee I' \le I} \qquad \frac{\Delta|\Gamma, x : I \vdash \mathcal{M} : \Theta}{\Delta|\Gamma \vdash \text{new}\{\, x : I; \mathcal{M}\,\} : I}\ {\scriptstyle \Delta(I) \upharpoonright Meths = \Theta}$$

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

IMJ: operational semantics

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

S stores object names + their types and values

Obj : set of obj. names

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

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

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

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

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

Program equivalence – decidability

$$M \cong M'$$

same observable behaviour in every context

for all closing contexts C:

$$(C[M],\phi) \rightarrow^* (\text{skip},S) \iff (C[M'],\phi) \rightarrow^* (\text{skip},S')$$

Program equivalence – decidability

$$M \cong M'$$

same observable behaviour in every context

for all closing contexts C:

$$(C[M],\phi) \to^* (\text{skip},S) \iff (C[M'],\phi) \to^* (\text{skip},S')$$

- we consider a finite-types restriction IMJ$_{fin}$
- termination is undecidable already in IMJ$_{fin}$ termination → non-equivalence
- we use game semantics to type-characterise the fragment of IMJ$_{fin}$ that is decidable for equivalence

Termination

M *terminates* if:
$(M,\phi) \rightarrow^* (\text{skip},S)$

Two distinct sources of undecidability:

I. recursive methods

$\text{new}(x : \mathcal{I};\ \text{main: } \lambda y.\ \{\ldots\ x.\text{main}(..)\ \ldots\ \})$

II. objects with methods can be stored

$\text{let } a = \text{new}(x : \mathcal{I};\ \text{main: } \lambda y.\ \{\ldots\ \})\ \text{ in}$
$\text{let } b = \text{new}(x' : \mathcal{I}';\ )\ \text{ in }\ b.\text{var} := a$

Theorem: Termination is undecidable for IMJ_{fin} terms that may feature I or II.
Conversely, given an IMJ_{fin} term M that does not feature I or II, we can always decide whether it terminates.

Games for program equivalence

We focus on: $\mathrm{IMJ}_{fin} - \{\mathrm{I},\mathrm{II}\}$

and employ the fully abstract game semantics for IMJ:

$$M \cong M' \Leftrightarrow [\![M]\!] = [\![M']\!]$$

- for negative results, given a queue machine Q, devise terms M, M':

$$Q\Uparrow \Leftrightarrow [\![M]\!] = [\![M']\!]$$

- for decidability, given terms M, M', devise automata A, A':

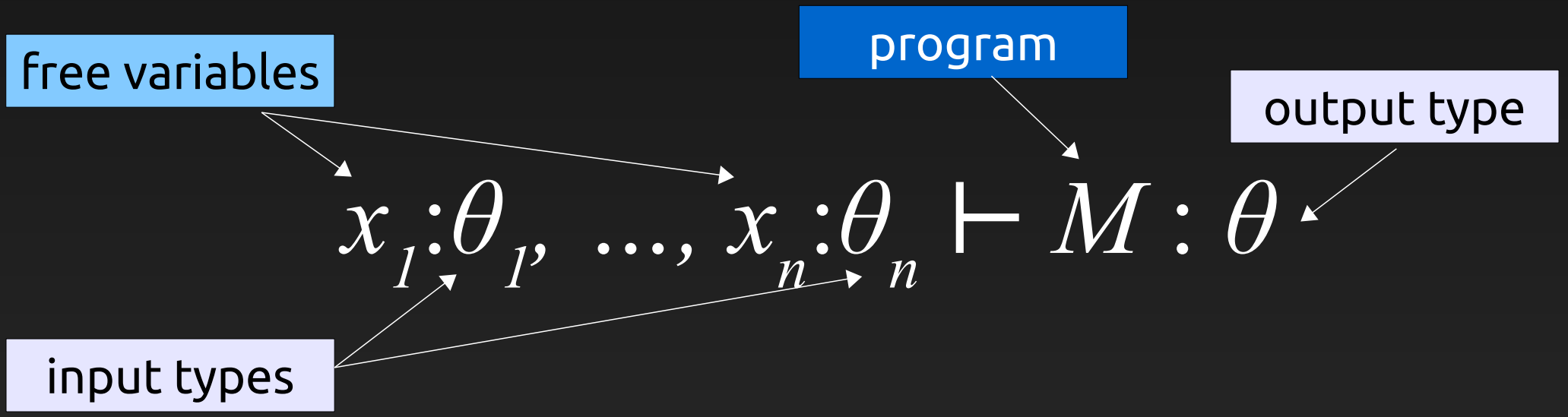
$$[\![M]\!] = [\![M']\!] \Leftrightarrow A \sim A'$$

Nominal game semantics

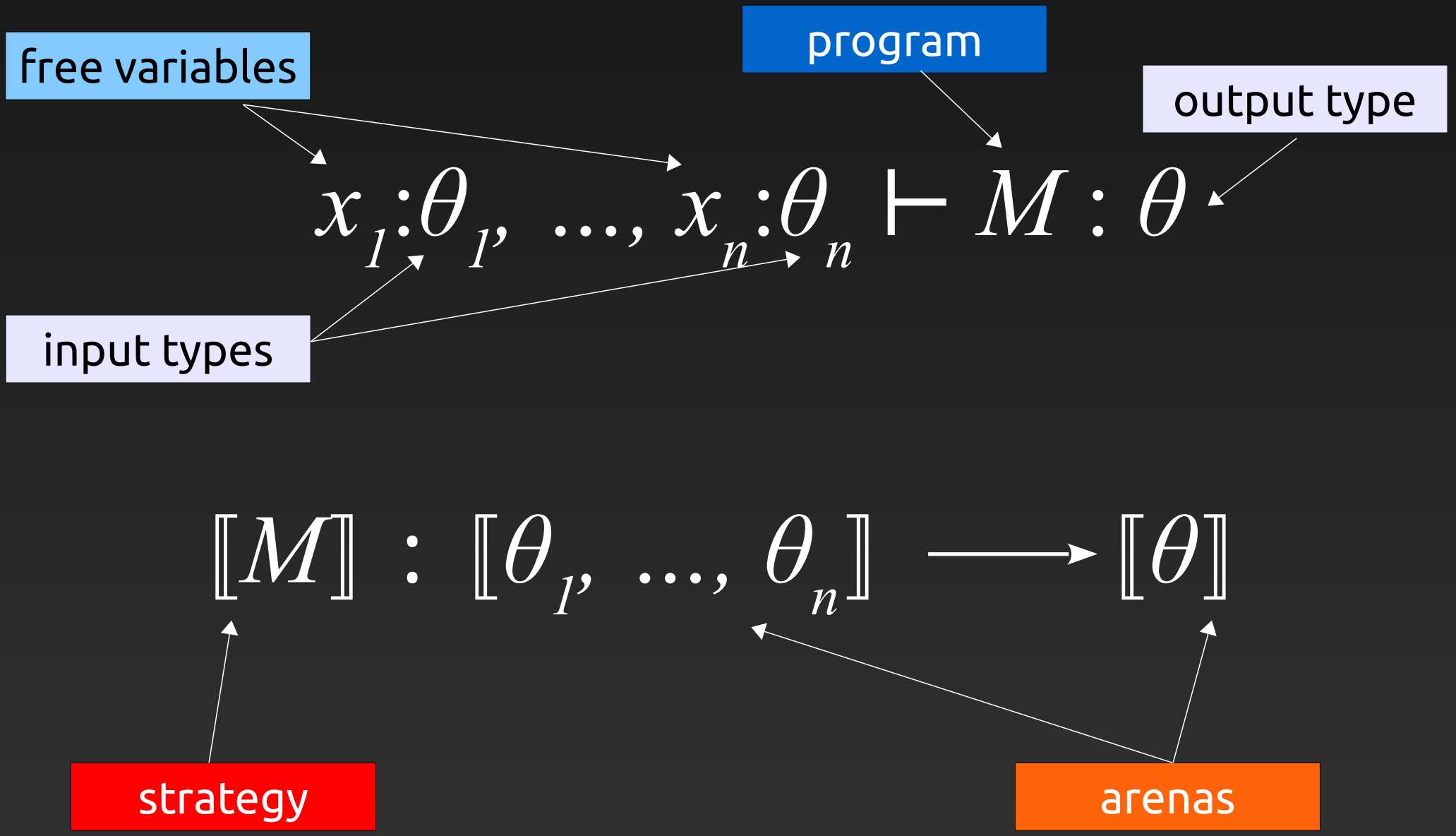
- Computation is a 2-player game between:
  - *Opponent* (the environment), aka O
  - *Proponent* (the program), aka P
- Programs = *strategies* for P
- *Categories* of games
- models cast in nominal sets
- games played using moves-with-stores
- special conditions for name privacy and propagation

[Gabbay & Pitts '02]

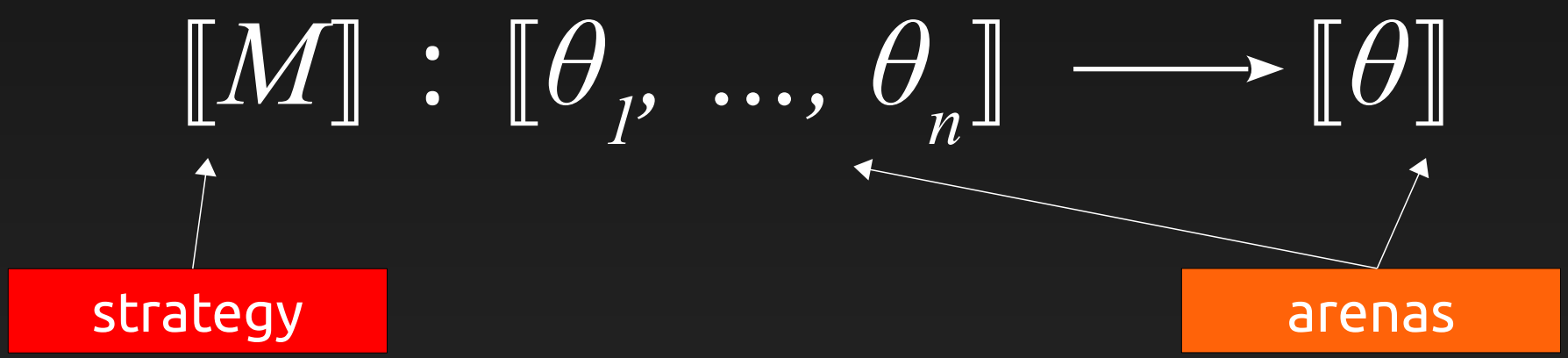
Arena games


free variables
program
output type
$x_1:\theta_1, \ldots, x_n:\theta_n \vdash M : \theta$
input types

Arena games


free variables
program
output type
$x_1:\theta_1, \ldots, x_n:\theta_n \vdash M : \theta$
input types
$[\![M]\!] : [\![\theta_1, \ldots, \theta_n]\!] \longrightarrow [\![\theta]\!]$
strategy
arenas

Arenas of moves



Arenas of moves

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

arenas

moves

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

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

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

$$a, b, \ldots \in \mathcal{N}_{\mathcal{I}}$$

$\mathcal{N}_{\mathcal{I}}$ a set of *names*

Arenas of moves

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

That's all!

arenas

moves

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

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

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

$a, b, \ldots \in \mathcal{N}_{\mathcal{I}}$

$\mathcal{N}_{\mathcal{I}}$ a set of *names*

Games for IMJ programs

- use names and moves-with-store:

$$\Sigma = \{\ (a,\ ),\ (b,\ (\mathsf{f}_1,4),\ (\mathsf{f}_2,c)),\ \ldots\ \}$$

- not really HO programs: no functions, only methods
  - *there is no need for "pointers" between moves*

effectively, moves of the form m^{Σ} where m is either:

- an arena move
- an object method call/return

[Murawski & T. '14]

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(Var_{Emp}) in
new(M_1) : Cell

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

Δ = Empty: ϕ,
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()$^{\Sigma_2}$ ...

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

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)

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

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

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

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)

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

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

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)

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

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

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)

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

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

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()$^{\Sigma_2}$ ... $= [\![M_2]\!]$

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

* Koutavas & Wand' 07

Games for program equivalence

We focus on: $\mathrm{IMJ}_{fin} - \{\mathrm{I},\mathrm{II}\}$

and employ the fully abstract game semantics for IMJ:

$$M \cong M' \Leftrightarrow [\![M]\!] = [\![M']\!]$$

- for negative results, given a queue machine Q, devise terms M, M':

$$Q\Uparrow \Leftrightarrow [\![M]\!] = [\![M']\!]$$

- for decidability, given terms M, M', devise automata A, A':

$$[\![M]\!] = [\![M']\!] \Leftrightarrow A \sim A'$$

Undecidability

One source of undecidability:

P passes objects of type $\mathcal{I}$, and $\mathcal{I}$ contains methods

$\mathcal{I}$: (step: void→void)
N : (val: int)

we can encode computations of queue machines!

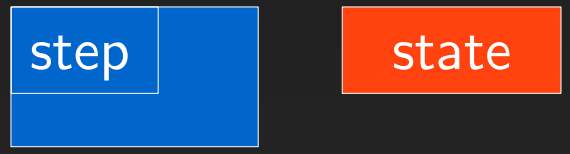
Undecidability

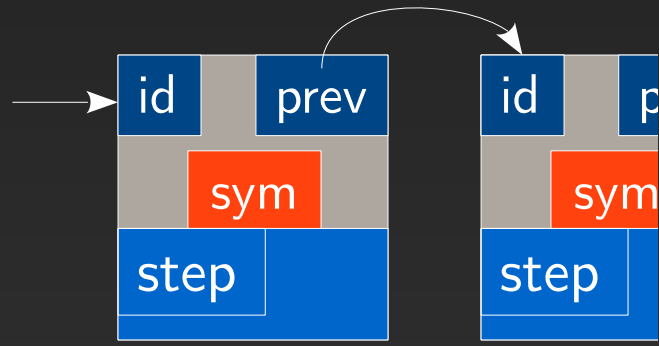
One source of undecidability:

P passes objects of type $\mathcal{I}$, and $\mathcal{I}$ contains methods

$\mathcal{I}$: (step: void→void)
N : (val: int)

we can encode computations of queue machines!





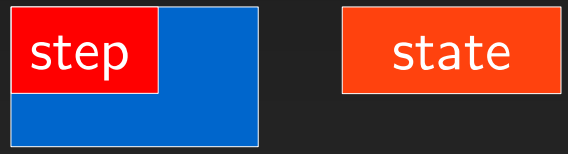
Undecidability

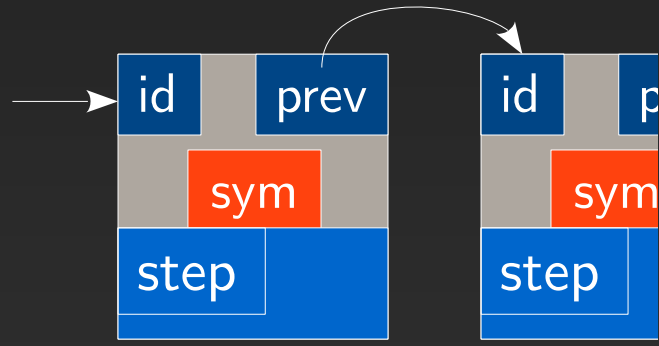
One source of undecidability:

P passes objects of type $\mathcal{I}$, and $\mathcal{I}$ contains methods

$\mathcal{I}$: (step: void→void)
N : (val: int)

we can encode computations of queue machines!





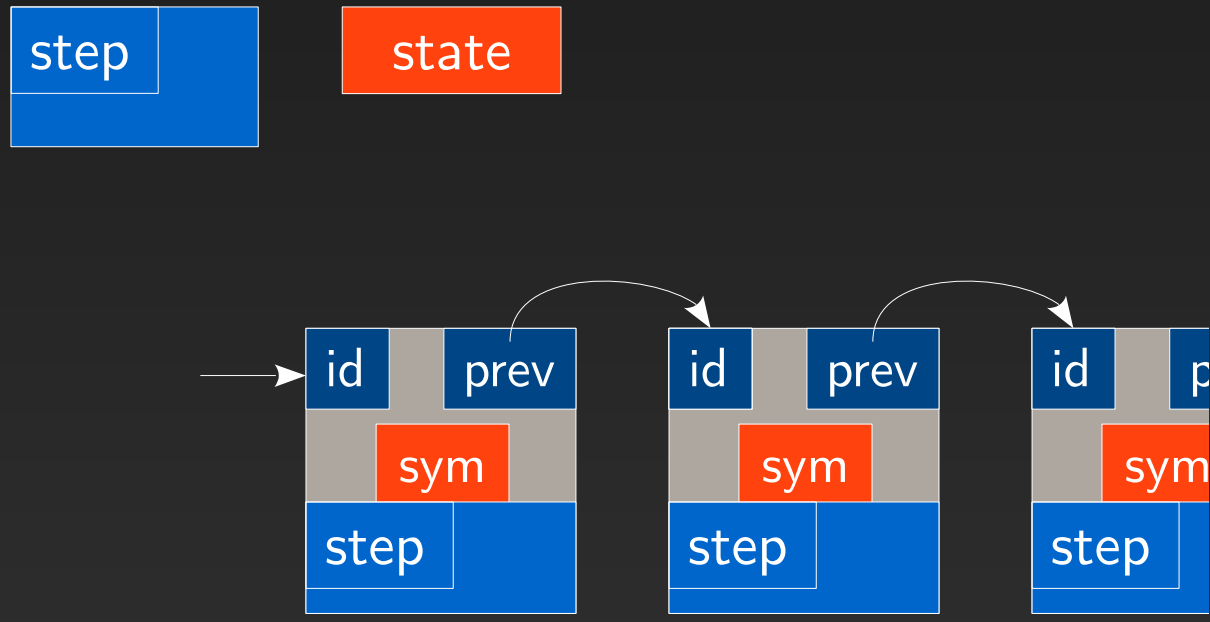
Undecidability

One source of undecidability:

P passes objects of type $\mathcal{I}$, and $\mathcal{I}$ contains methods

$\mathcal{I}$: (step: void→void)
N : (val: int)

we can encode computations of queue machines!



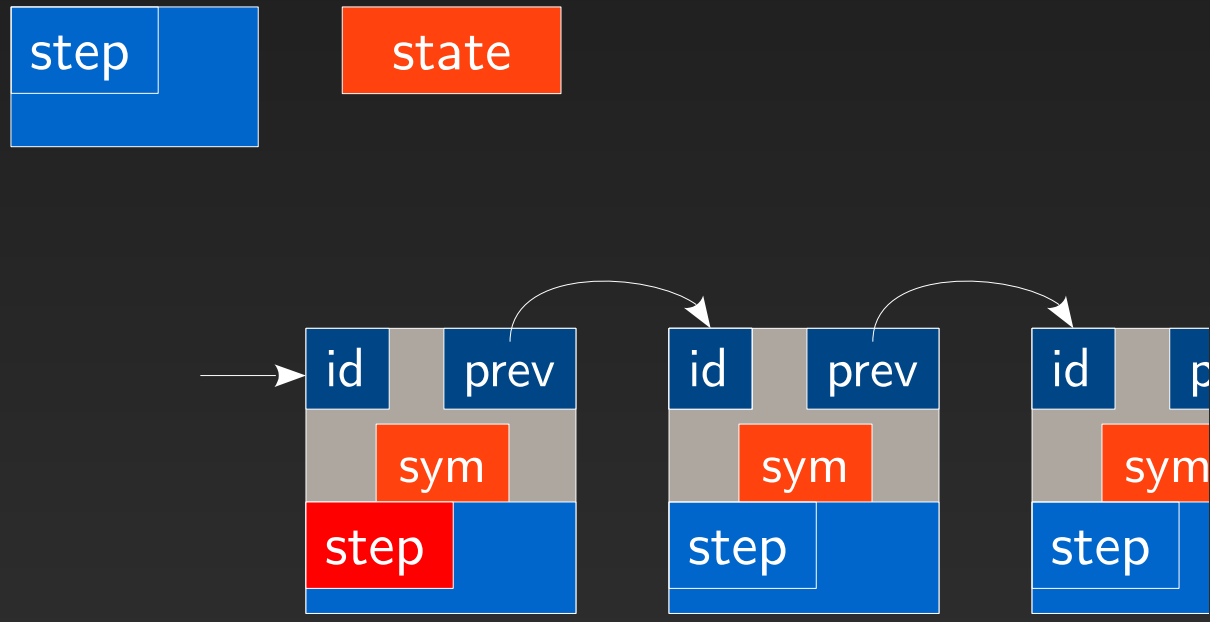
Undecidability

One source of undecidability:

P passes objects of type $\mathcal{I}$, and $\mathcal{I}$ contains methods

$\mathcal{I}$: (step: void→void)
N : (val: int)

we can encode computations of queue machines!



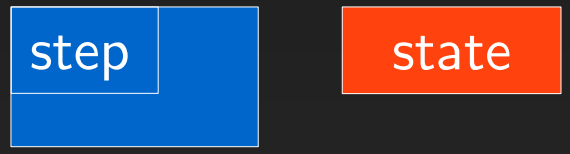
Undecidability

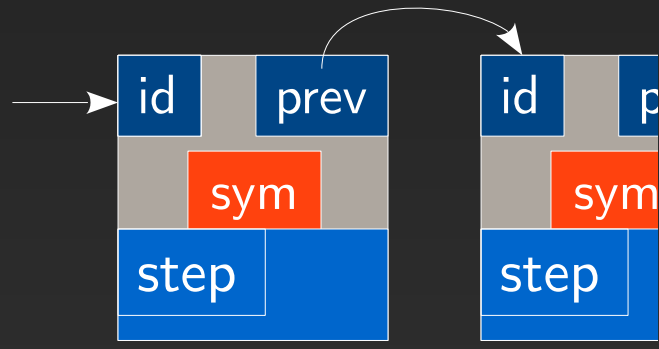
One source of undecidability:

P passes objects of type $\mathcal{I}$, and $\mathcal{I}$ contains methods

$\mathcal{I}$: (step: void→void)
N : (val: int)

we can encode computations of queue machines!





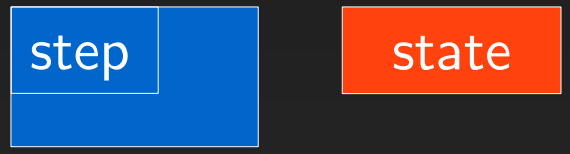
Undecidability

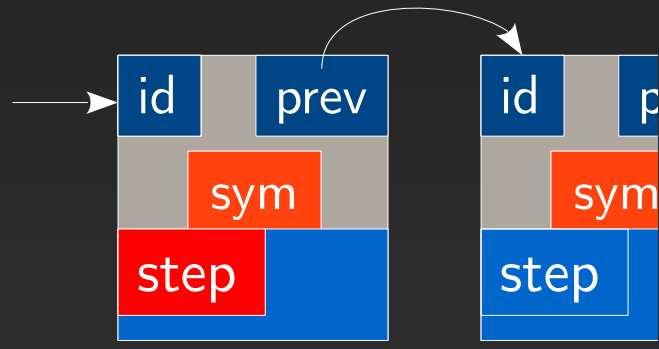
One source of undecidability:

P passes objects of type $\mathcal{I}$, and $\mathcal{I}$ contains methods

$\mathcal{I}$: (step: void→void)
N : (val: int)

we can encode computations of queue machines!





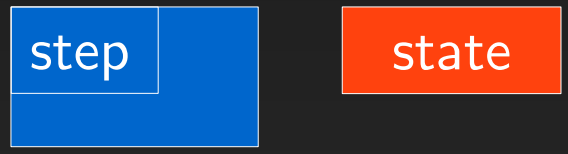
Undecidability

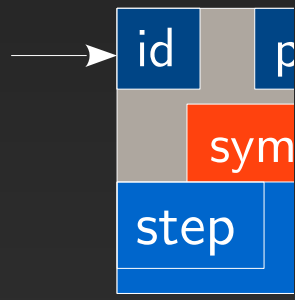
One source of undecidability:

P passes objects of type $\mathcal{I}$, and $\mathcal{I}$ contains methods

$\mathcal{I}$: (step: void→void)
N : (val: int)

we can encode computations of queue machines!





Undecidability

$$x : \mathsf{L} \vdash M : \mathsf{R}$$

Three cases for undecidability:

- $\mathsf{L} = (..., \mathsf{m} : \mathcal{I} \to \theta)$
- $\mathsf{R} = (..., \mathsf{m} : \theta \to \mathcal{I})$
- $\mathsf{R} = (..., \mathsf{m} : \mathcal{I}' \to \theta)$ where $\mathcal{I}' = (..., \mathsf{m} : \mathcal{I} \to \theta)$

and $\mathcal{I}$ contains methods (i.e. is higher-order)

Decidable types

$x : \mathsf{L} \vdash M : \mathsf{R}$

$$\mathsf{G} \;::=\; \mathsf{void} \mid \mathsf{int} \mid (\mathsf{f} : \mathsf{G})$$

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

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

Games for program equivalence

We focus on: $\mathrm{IMJ}_{fin} - \{\mathrm{I},\mathrm{II}\}$

and employ the fully abstract game semantics for IMJ:

$$M \cong M' \Leftrightarrow [\![M]\!] = [\![M']\!]$$

- for negative results, given a queue machine Q, devise terms M, M':

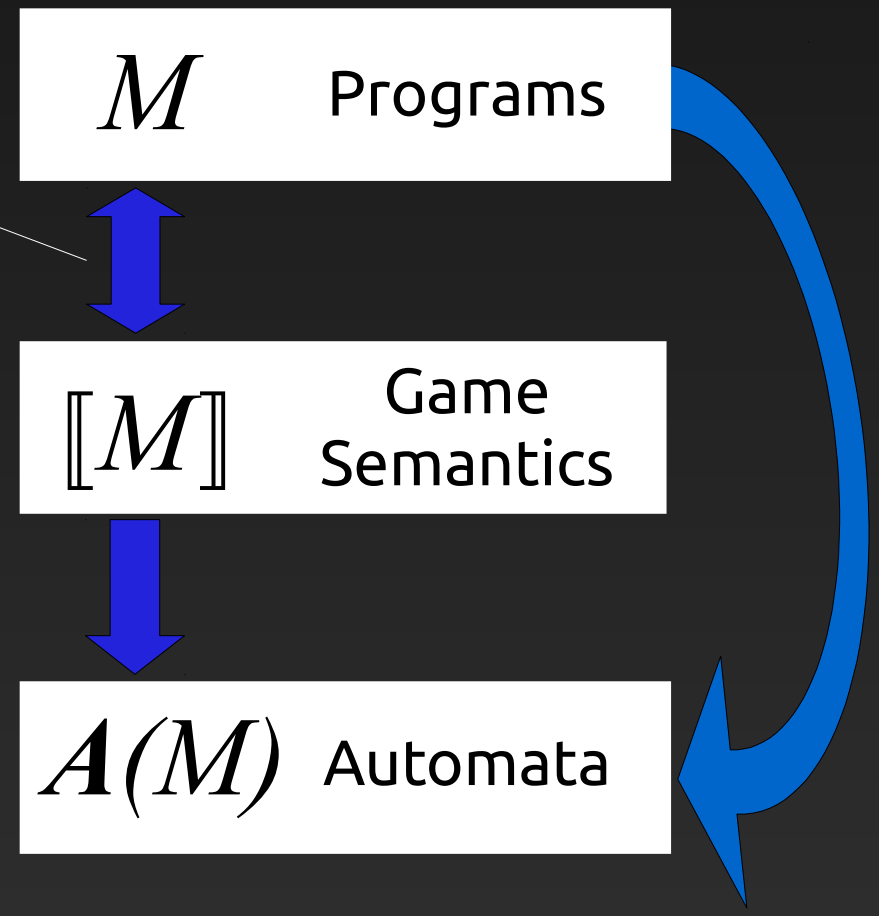
$$Q\Uparrow \Leftrightarrow [\![M]\!] = [\![M']\!]$$

- for decidability, given terms M, M', devise automata A, A':

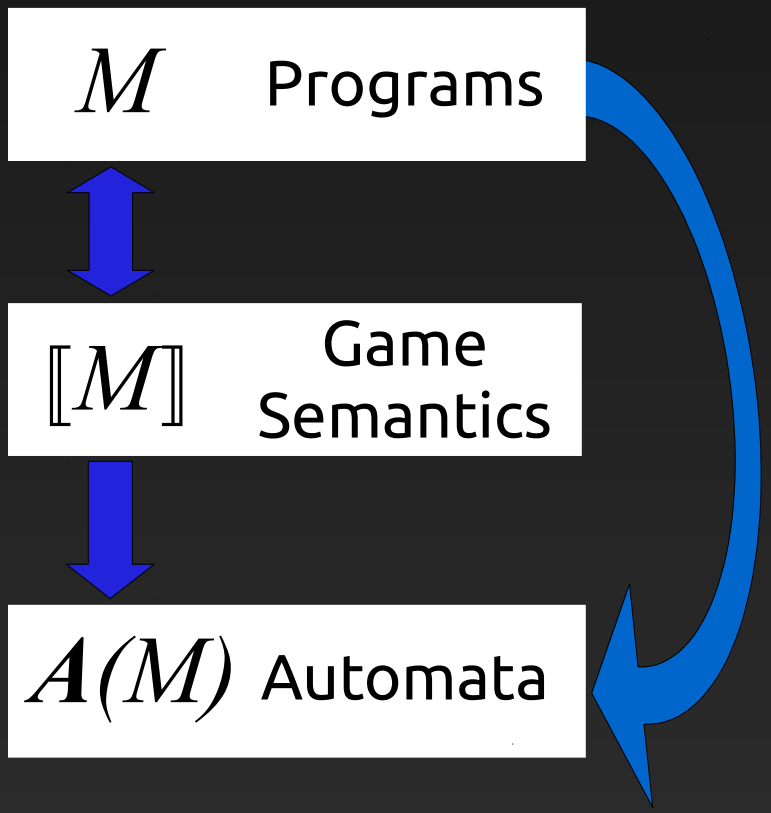
$$[\![M]\!] = [\![M']\!] \Leftrightarrow A \sim A'$$

From games to automata

$$M \cong M'$$
$$\Leftrightarrow$$
$$[\![M]\!] = [\![M']\!]$$



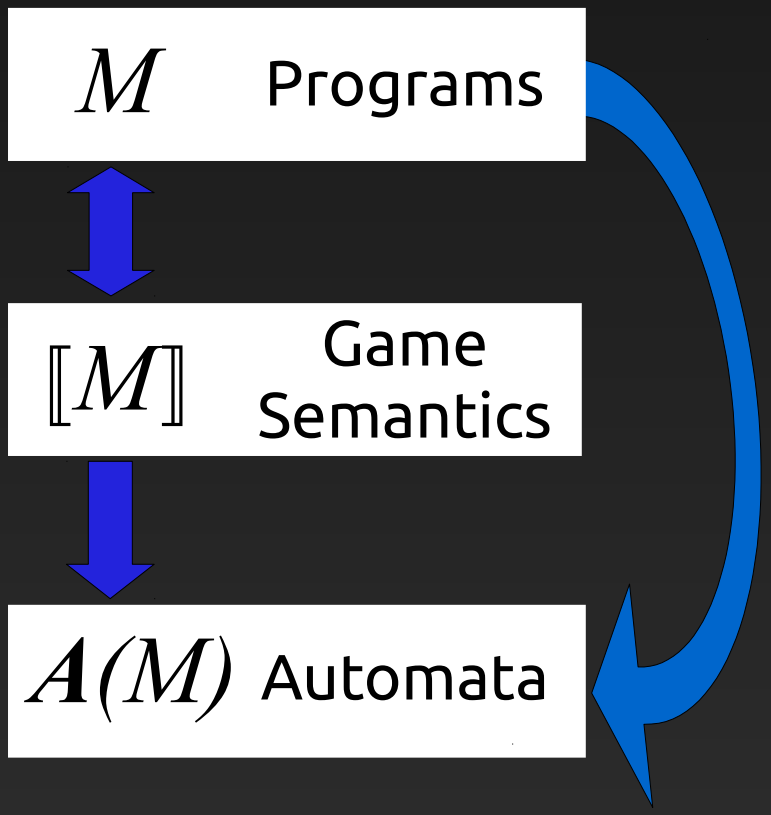
From games to automata



Translation done in two steps:

- terms reduced to canonical forms
- canonical terms are compositionally transformed into automata

From games to automata



Translation done in two steps:

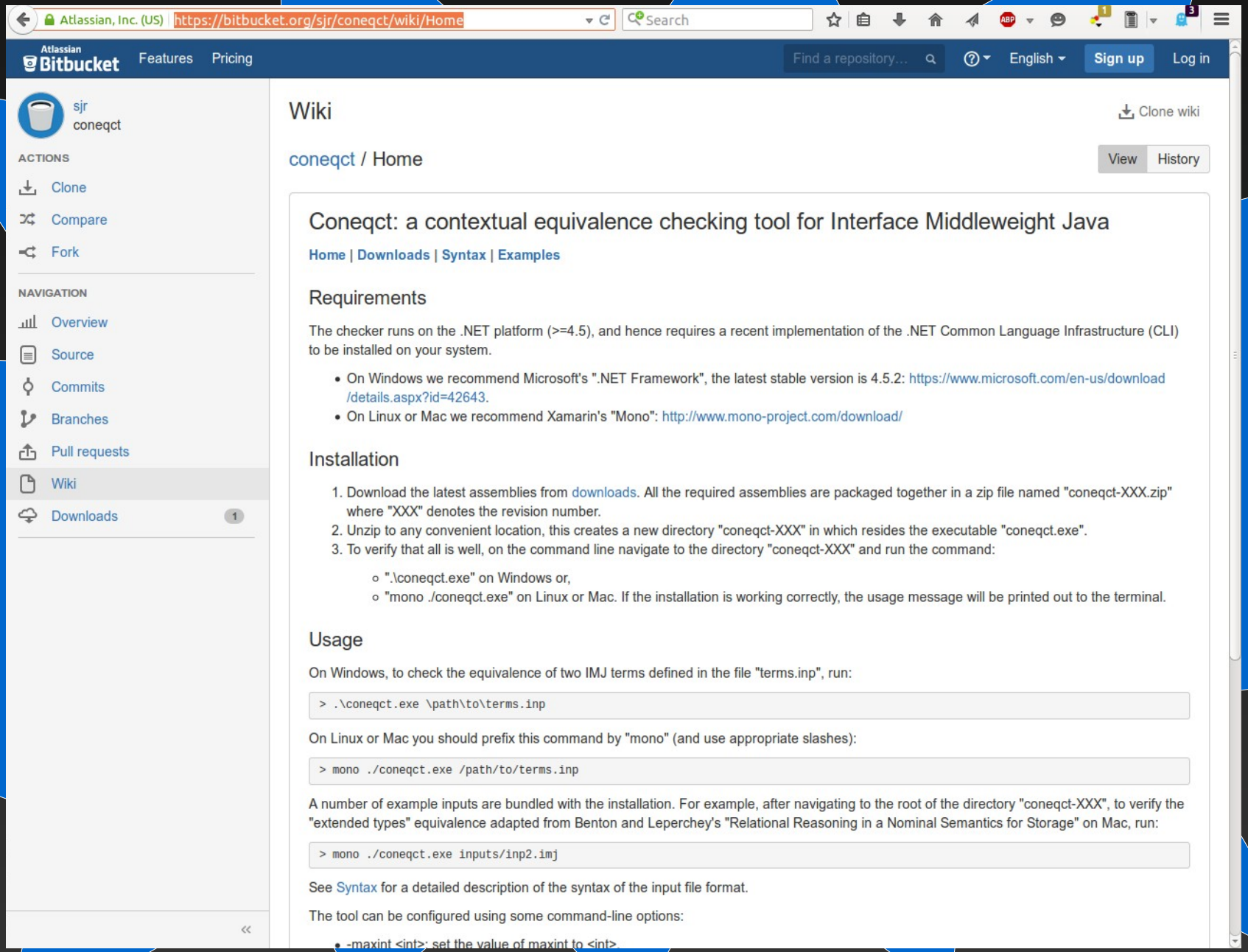
- terms reduced to canonical forms
- canonical terms are compositionally transformed into automata

We work with pushdown automata:

- over infinite alphabets
- visibly pushdown & deterministic

$$M \cong M' \Leftrightarrow [\![M]\!] = [\![M']\!] \Leftrightarrow A \sim A' \Leftrightarrow A \otimes A' = 0$$

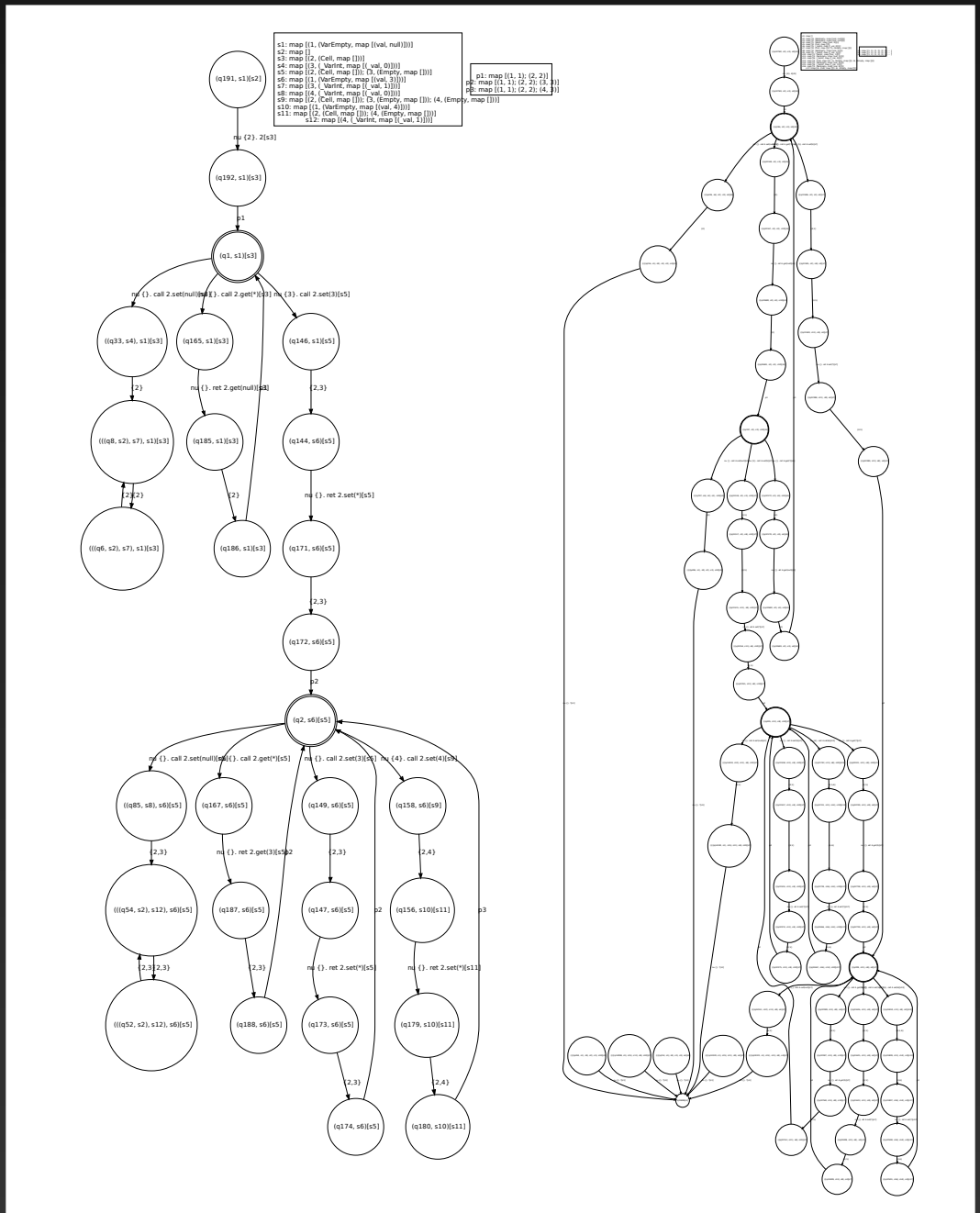
Coneqct


Atlassian, Inc. (US) https://bitbucket.org/sjr/coneqct/wiki/Home
Search
Atlassian Bitbucket
Features
Pricing
Find a repository...
English
Sign up
Log in
sjr
coneqct
ACTIONS
Clone
Compare
Fork
NAVIGATION
Overview
Source
Commits
Branches
Pull requests
Wiki
Downloads 1
Wiki
Clone wiki
coneqct / Home
View
History
Coneqct: a contextual equivalence checking tool for Interface Middleweight Java
Home | Downloads | Syntax | Examples
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 Windows we recommend Microsoft's ".NET Framework", the latest stable version is 4.5.2: https://www.microsoft.com/en-us/download/details.aspx?id=42643.
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 "coneqct-XXX.zip" where "XXX" denotes the revision number.
2. Unzip to any convenient location, this creates a new directory "coneqct-XXX" in which resides the executable "coneqct.exe".
3. To verify that all is well, on the command line navigate to the directory "coneqct-XXX" and run the command:
"\coneqct.exe" on Windows or,
"mono ./coneqct.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:
> .\coneqct.exe \path\to\terms.inp
On Linux or Mac you should prefix this command by "mono" (and use appropriate slashes):
> mono ./coneqct.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 "coneqct-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 ./coneqct.exe inputs/inp2.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>.

Coneqct

$$
\begin{aligned}
M_1 \equiv\ & \text{let } v = \text{new } \{\ \_\colon \mathsf{Var}_{\mathsf{Empty}}\} \text{ in} \\
& \text{new } \{\ \_\colon \mathsf{Cell}; \\
& \quad get\ :\ \lambda\_.\ v.val, \\
& \quad set\ :\ \lambda y.\ \text{if } y{=}\text{null then div else } v.val{:=}y\ \}
\end{aligned}
$$

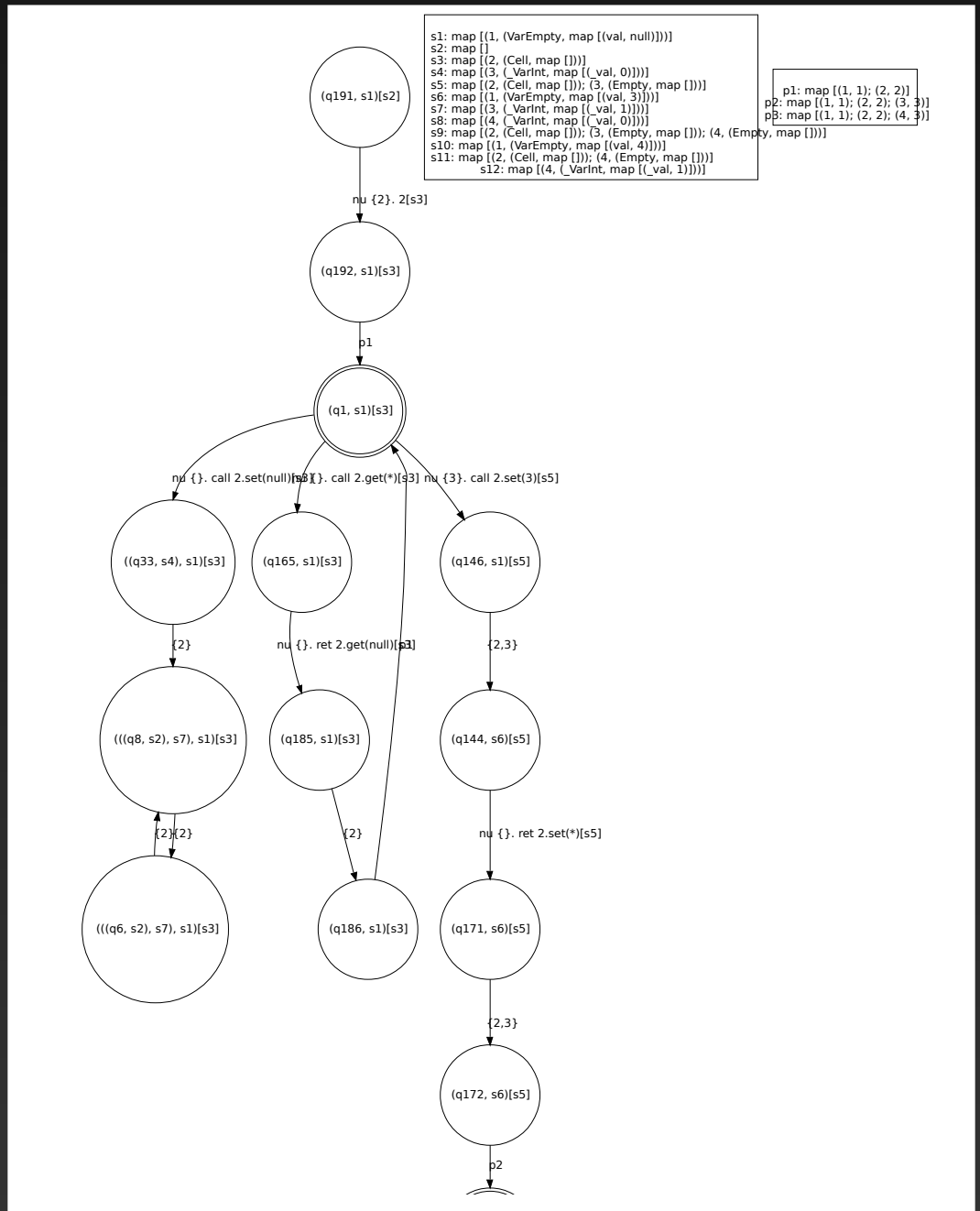
$$
\begin{aligned}
M_2 \equiv\ & \text{let } b = \text{new } \{\ \_\colon \mathsf{Var}_{\mathsf{int}}\} \text{ in} \\
& \text{let } v = \text{new } \{\ \_\colon \mathsf{Var}_{\mathsf{Empty}}\} \text{ in} \\
& \text{let } w = \text{new } \{\ \_\colon \mathsf{Var}_{\mathsf{Empty}}\} \text{ in} \\
& \text{new } \{\ \_\colon \mathsf{Cell}; \\
& \quad get\ :\ \lambda\_.\ \text{if } b.val{=}1 \text{ then } (b.val{:=}0;\ v.val) \\
& \qquad\qquad \text{else } (b.val{:=}1;\ w.val), \\
& \quad set\ :\ \lambda y.\ \text{if } y{=}\text{null then div} \\
& \qquad\qquad \text{else } v.val{:=}y;\ w.val{:=}y\ \}
\end{aligned}
$$



Coneqct

$$M_1 \equiv \text{let } v = \text{new } \{\ \_\colon \mathsf{Var}_{\mathsf{Empty}}\} \text{ in}$$
$$\text{new } \{\ \_\colon \text{Cell};$$
$$get\ :\ \lambda\_.\ v.val,$$
$$set\ :\ \lambda y.\ \text{if } y{=}\text{null then div else } v.val{:=}y\ \}$$

$$M_2 \equiv \text{let } b = \text{new } \{\ \_\colon \mathsf{Var}_{\mathsf{int}}\} \text{ in}$$
$$\text{let } v = \text{new } \{\ \_\colon \mathsf{Var}_{\mathsf{Empty}}\} \text{ in}$$
$$\text{let } w = \text{new } \{\ \_\colon \mathsf{Var}_{\mathsf{Empty}}\} \text{ in}$$
$$\text{new } \{\ \_\colon \text{Cell};$$
$$get\ :\ \lambda\_.\ \text{if } b.val{=}1 \text{ then } (b.val{:=}0;\ v.val)$$
$$\text{else } (b.val{:=}1;\ w.val),$$
$$set\ :\ \lambda y.\ \text{if } y{=}\text{null then div}$$
$$\text{else } v.val{:=}y;\ w.val{:=}y\ \}$$



Wrapping up

Games for IMJ:

- are simple and operational
- used here for characterising program equivalence
- automated decision via FPDRAs → Coneqct

Further on:

- general model checking
- logics (over infinite alphabets)

Wrapping up

Games for IMJ:

- are simple and operational
- used here for characterising program equivalence
- automated decision via FPDRAs → Coneqct

Further on:

- general model checking
- logics (over infinite alphabets)

the End

thanks!