Model Checking Program Equivalence in Interface Middleweight Java

Andrzej Murawski
Uni. of Warwick

Steven Ramsay
Uni. of Oxford

Nikos Tzevelekos
Queen Mary U. of London

PPLV seminar, UCL, Apr 2016

Supported by a Royal Academy of Engineering Research Fellowship

what this talk is about

We examine contextual equivalence of Java programs

We work in an fragment of Middleweight Java

- open code modelled by use of interfaces
- denotational approach based on game semantics

Full characterisation based on type disciplines:

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

Implementation of decidable fragment in Coneqct

Program equivalence

$$M \cong M'$$

same observable behaviour in every context

for all closing contexts $C[]$:

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

- subsumes standard verification properties
- contextual: quantification over *every* context
- several methods for proving given equivalences yet no general procedures

Equivalent?

$M_1 \equiv$ let $x =$ new(VarInt) in 0 : int

$M_2 \equiv$ 0 : int

VarInt: { val: int }

Equivalent?

$M_1 \equiv$ let $x =$ new(VarInt) in 0 : int

$M_2 \equiv$ 0 : int

VarInt: { val: int }

$M_1 \equiv$ let $x =$ new(VarInt) in
y.val := x; 0 : int

$M_2 \equiv$ 0 : int

VarInt: { val: int },
VarVarInt: { val: VarInt }
y: VarVarInt

Equivalent?

$M_1 \equiv$ let $x =$ new(VarInt) in 0 : int

$M_2 \equiv$ 0 : int

VarInt: { val: int }

$M_1 \equiv$ let $x =$ new(VarInt) in
y.val := x; 0 : int

$M_2 \equiv$ 0 : int

VarInt: { val: int },
VarVarInt: { val: VarInt }
y: VarVarInt

$C[] \equiv$ let $y =$ new(VarVarInt) in
let _ = [] in y.val.val := 42 : void

Interface Middleweight Java (IMJ)

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

Interface definitions

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

Interface tables

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

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

- Objects, inheritance, casting, interfaces

Interface Middleweight Java (IMJ)

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

Interface definitions

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

Interface tables

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

interface ident.

field identifier

method identif.

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

- Objects, inheritance, casting, interfaces

Interface Middleweight Java (IMJ)

Terms

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

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

$$\frac{(x : \theta) \in \Gamma}{\Delta \mid \Gamma \vdash x : \theta} \qquad \frac{(a : \mathcal{I}) \in \Gamma}{\Delta \mid \Gamma \vdash a : \mathcal{I}} \qquad \frac{\Delta(\mathcal{I}).\mathsf{f} = \theta \quad \Delta \mid \Gamma \vdash M : \mathcal{I}}{\Delta \mid \Gamma \vdash M.\mathsf{f} : \theta}$$

$$\frac{}{\Delta \mid \Gamma \vdash \text{null} : \mathcal{I}} \qquad \frac{\Delta \mid \Gamma, x : \mathcal{I} \vdash \{\mathsf{m}_i : \lambda\overline{x}_i.M_i : \overline{\theta}_i \to \theta_i\}}{\Delta \mid \Gamma \vdash \text{new}(x : \mathcal{I};\ \{\mathsf{m}_i : \lambda\overline{x}_i.M_i\}) : \mathcal{I}}$$

IMJ: operational semantics

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

S stores object names + their types and values

Obj : set of obj. names

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

$$S,\ a = a' \ \longrightarrow\ S,\ 0/1$$

$a,a' \in \text{Obj}$

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

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

Equivalent?

```
VarInt = { val: int },
I = { run: void -> void },
f: I
-----------------------------------------------

let x = new {_: VarInt;} in
new {_:I;
  run: \_. if x.val=0 then (
                  x.val:= 1; f.run();
                  if x.val = 2 then skip else div
                )
                else (
                  if x.val = 1 then x.val:= 2 else div
                )
}
  =?=

new {_:I; run:\_. div}
```

Program equivalence – the plan

$$M \cong M'$$

same observable behaviour in every context

for all closing contexts C:

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

Program equivalence – the plan

$$M \cong M'$$

same observable behaviour in every context

for all closing contexts C:

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

termination → inequivalence

we consider a finitary restriction IMJ_{fin} and first identify the fragment decidable for termination

we use game semantics and types to characterise the fragment of IMJ_{fin} 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

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

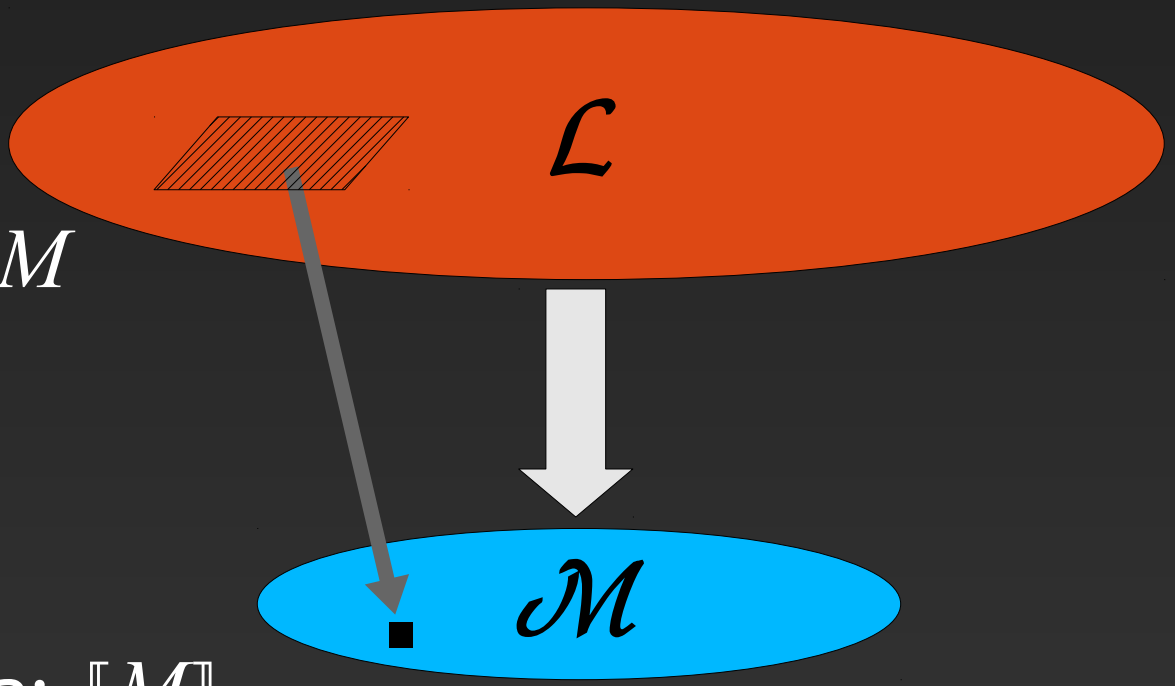
Theorem: Termination is undecidable for IMJ_{fin} terms that may feature I or II.
Conversely, given an IMJ_{fin} term M that does not feature I or II, we can always decide whether it terminates.

Games for program equivalence

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

and employ the fully abstract game semantics for IMJ:

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



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

Model the execution of programs as a 2-player game:

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

Qualitative games (no winning conditions)

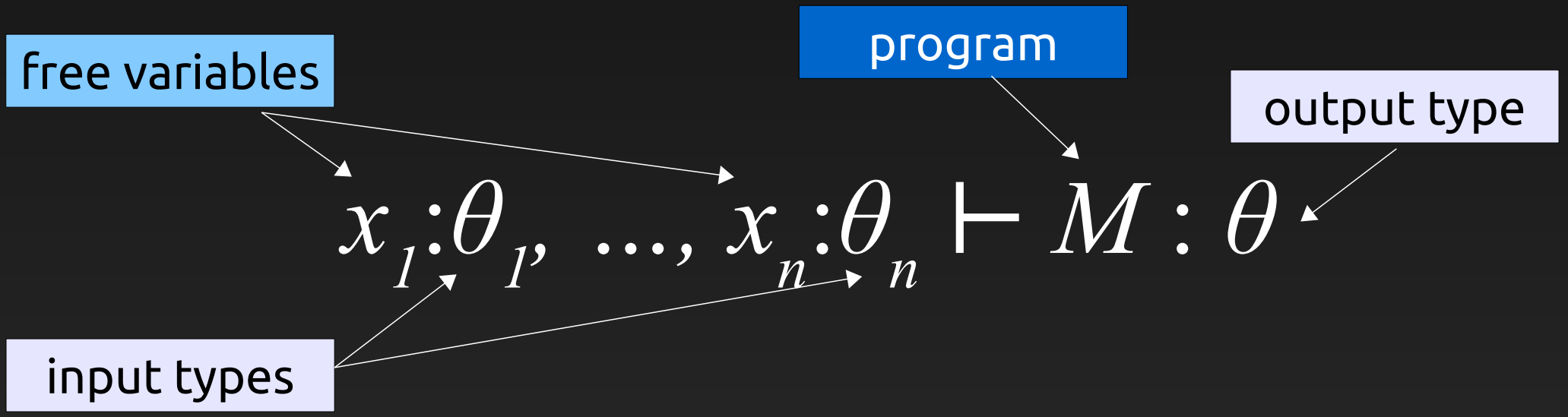
Computations = *plays* of a specified game

Programs = *strategies* for P

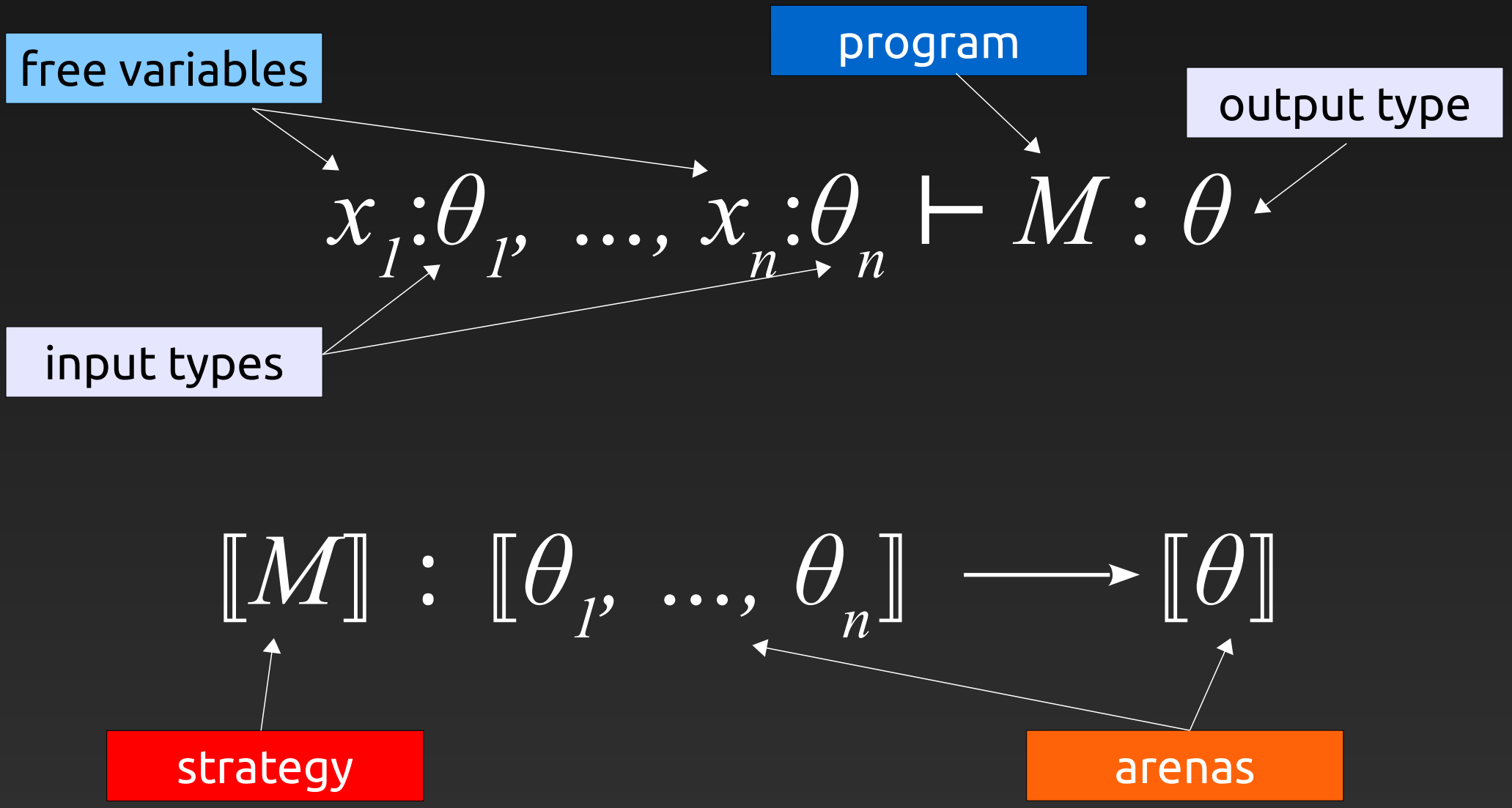
Categories of games, extended with names & effects

- storage: moves with stores
- conditions for name privacy and propagation

Game apparatus



Game apparatus



Initial game moves

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

strategy

arenas

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

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

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

moves

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

Games for IMJ programs

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

- move m is either:
  - a value/arena move
  - an object method call/return

call c.set(c'), ret q.get(), …

- stores are of the form:

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

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

[Murawski & T. '14]

IMJ example: game semantics

$M_1 \equiv$ let $u =$ new(VarEmp) in
new(M_1) : Cell

$M_1 \equiv$ get: $\lambda\_.\ u.\mathsf{val}$,
set: $\lambda y.\ u.\mathsf{val} := y$

$\Delta =$ Empty: $\emptyset$,
Cell: (get: void $\rightarrow$ Empty,
set: Empty $\rightarrow$ void),
VarEmp: (val: Empty),
VarInt: (val: int)

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

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

* [Koutavas & Wand' 07]

IMJ example: game semantics

$M_2 \equiv$ let $b =$ new(VarInt) in
let $u_1 =$ new(VarEmp) in
let $u_2 =$ new(VarEmp) in
new(M_2) : Cell

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

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

$[\![M_1]\!] =$ $*$ c^{Σ_0} (call c.get()$^{\Sigma_0}$ ret c.get(nul)$^{\Sigma_0}$)*
call c.set(n_1)$^{\Sigma_1}$ ret c.set()$^{\Sigma_1}$
(call c.get()$^{\Sigma_1}$ ret c.get(n_1)$^{\Sigma_1}$)*
call c.set(n_2)$^{\Sigma_2}$ ret c.set()$^{\Sigma_2}$ ... $= [\![M_2]\!]$

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

* [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':

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

One source of Undecidability

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

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

(run: $\mathcal{I}\rightarrow$void) $\vdash$ void

we can encode computations of queue machines!

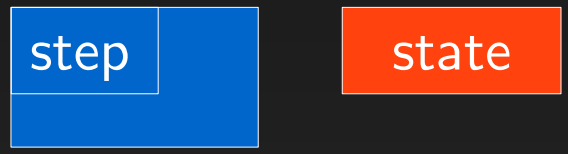
One source of Undecidability

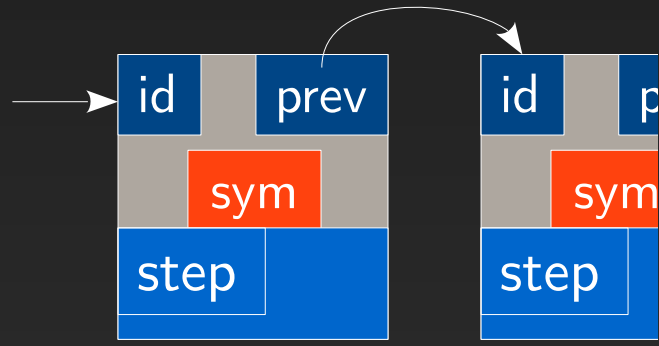
P passes objects of type $\mathcal{I}$, and $\mathcal{I}$ contains methods

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

(run: $\mathcal{I}$→void) ⊢ void

we can encode computations of queue machines!





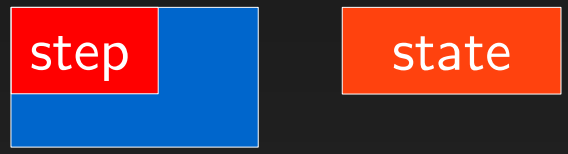
One source of Undecidability

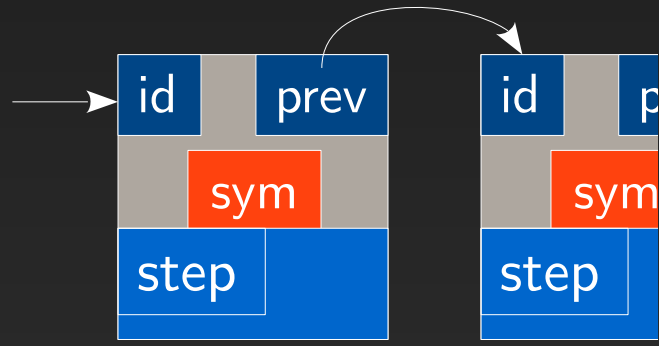
P passes objects of type $\mathcal{I}$, and $\mathcal{I}$ contains methods

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

(run: $\mathcal{I}$→void) ⊢ void

we can encode computations of queue machines!





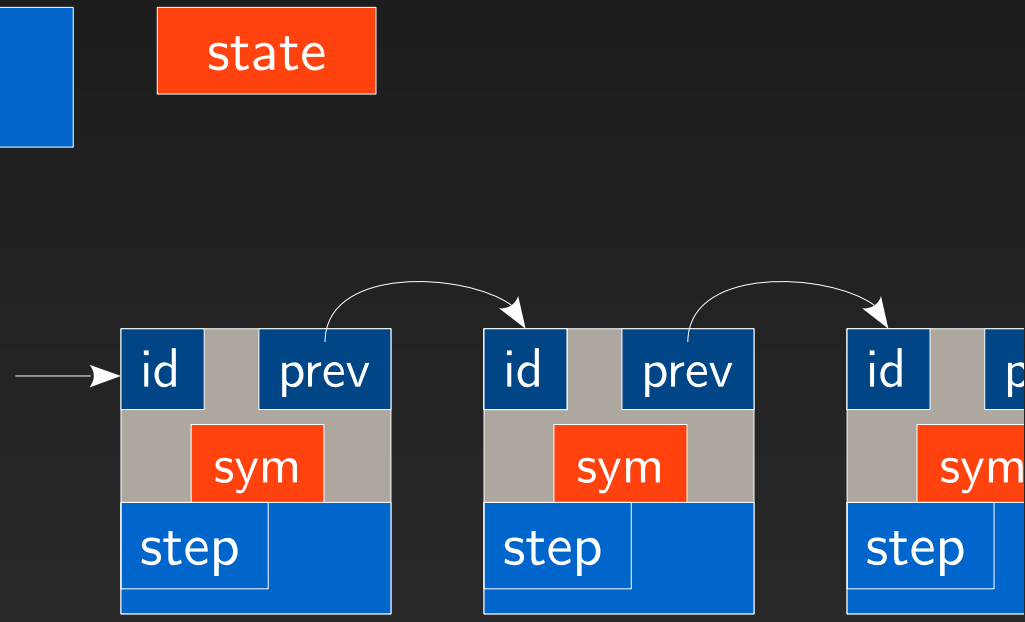
One source of Undecidability

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

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

(run: $\mathcal{I}$→void) ⊢ void

we can encode computations of queue machines!



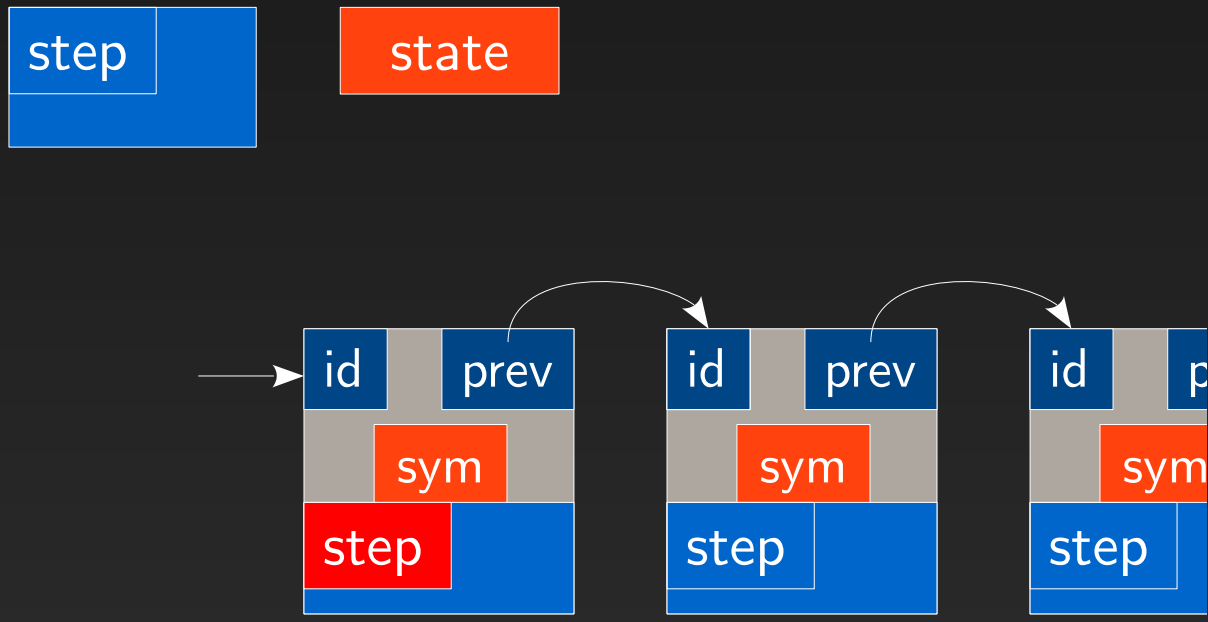
One source of Undecidability

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

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

(run: $\mathcal{I}$→void) ⊢ void

we can encode computations of queue machines!



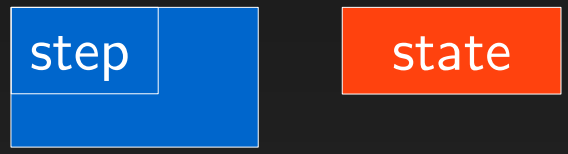
One source of Undecidability

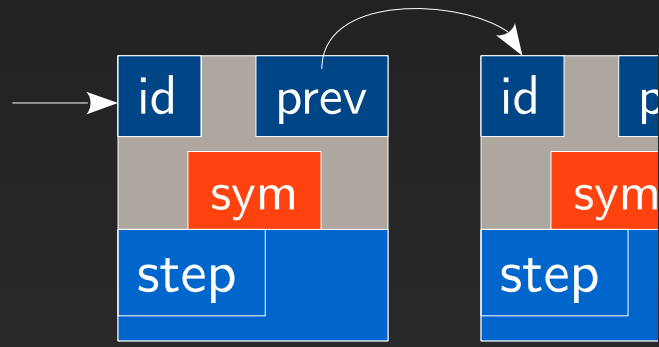
P passes objects of type $\mathcal{I}$, and $\mathcal{I}$ contains methods

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

(run: $\mathcal{I}$→void) ⊢ void

we can encode computations of queue machines!





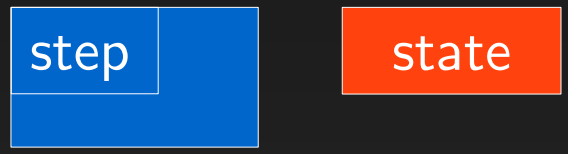
One source of Undecidability

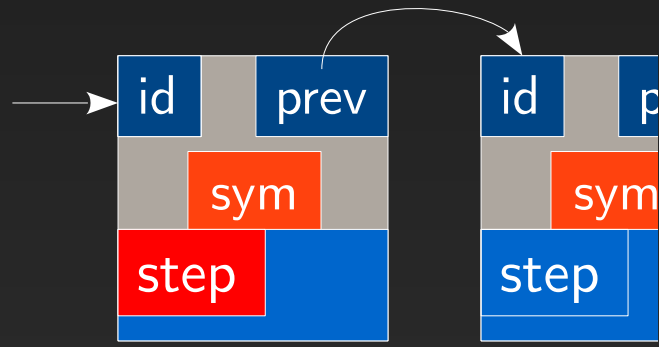
P passes objects of type $\mathcal{I}$, and $\mathcal{I}$ contains methods

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

(run: $\mathcal{I}$→void) ⊢ void

we can encode computations of queue machines!





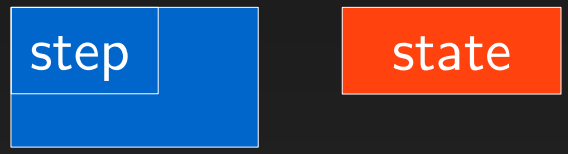
One source of Undecidability

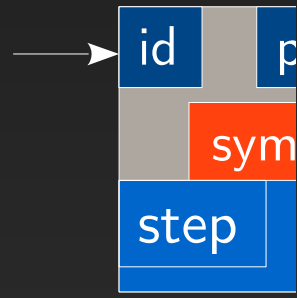
P passes objects of type $\mathcal{I}$, and $\mathcal{I}$ contains methods

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

(run: $\mathcal{I}$→void) ⊢ void

we can encode computations of queue machines!





Undecidability

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

Three cases for undecidability:

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

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

Decidable types – IMJ*

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

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

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

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

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':

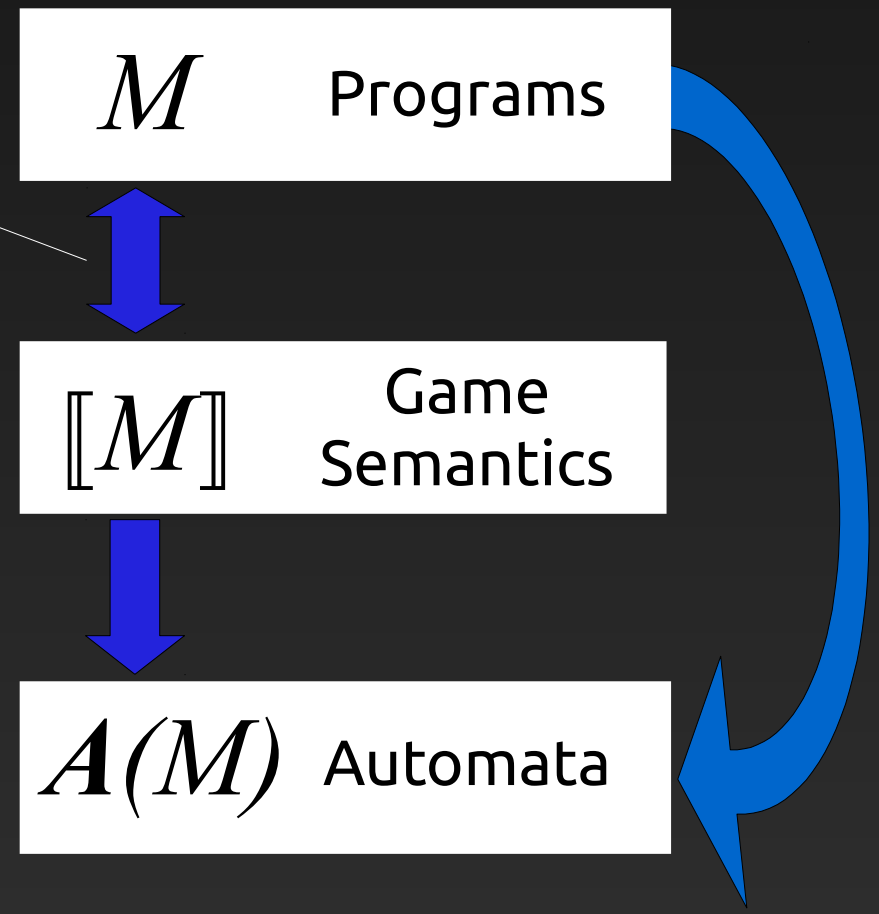
$$Q\Uparrow \Leftrightarrow [\![M]\!] = [\![M']\!]$$

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

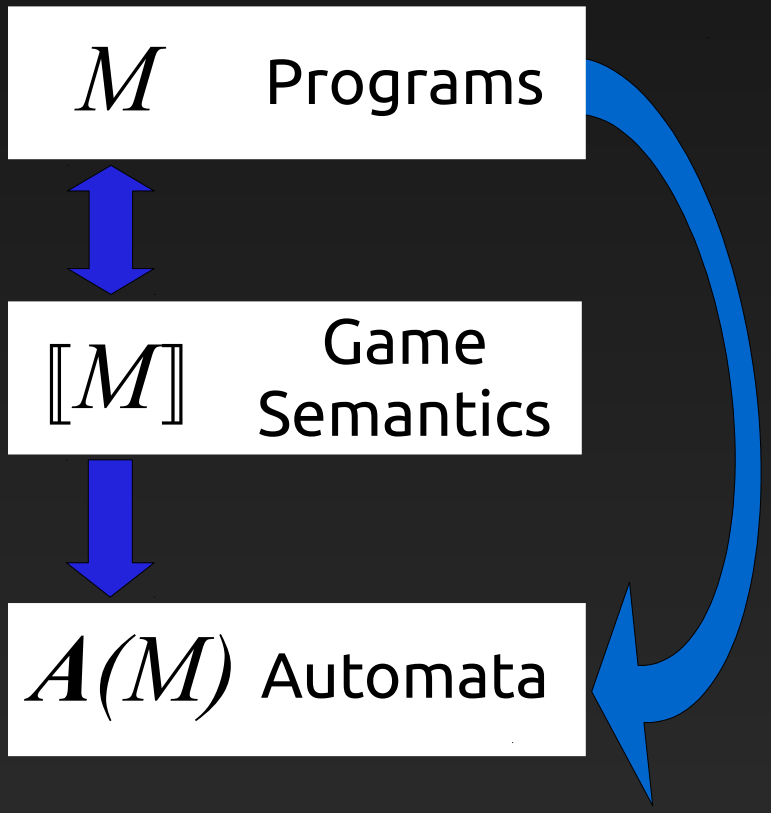
$$[\![M]\!] = [\![M']\!] \Leftrightarrow A \sim A'$$

Model checking equivalence

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



Model checking equivalence



Translation done in two steps:

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

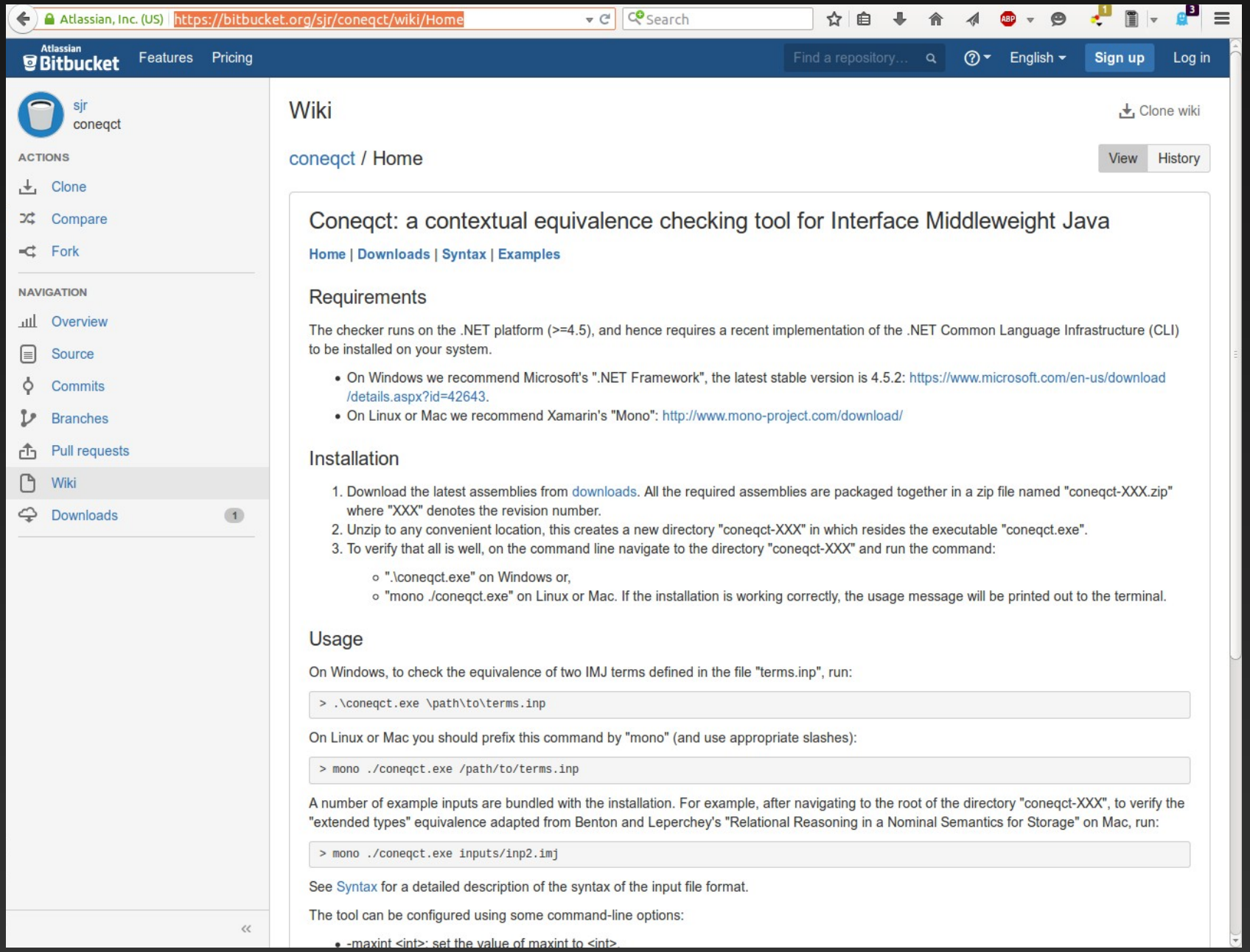
We work with pushdown automata:

- over infinite alphabets
- visibly pushdown & deterministic

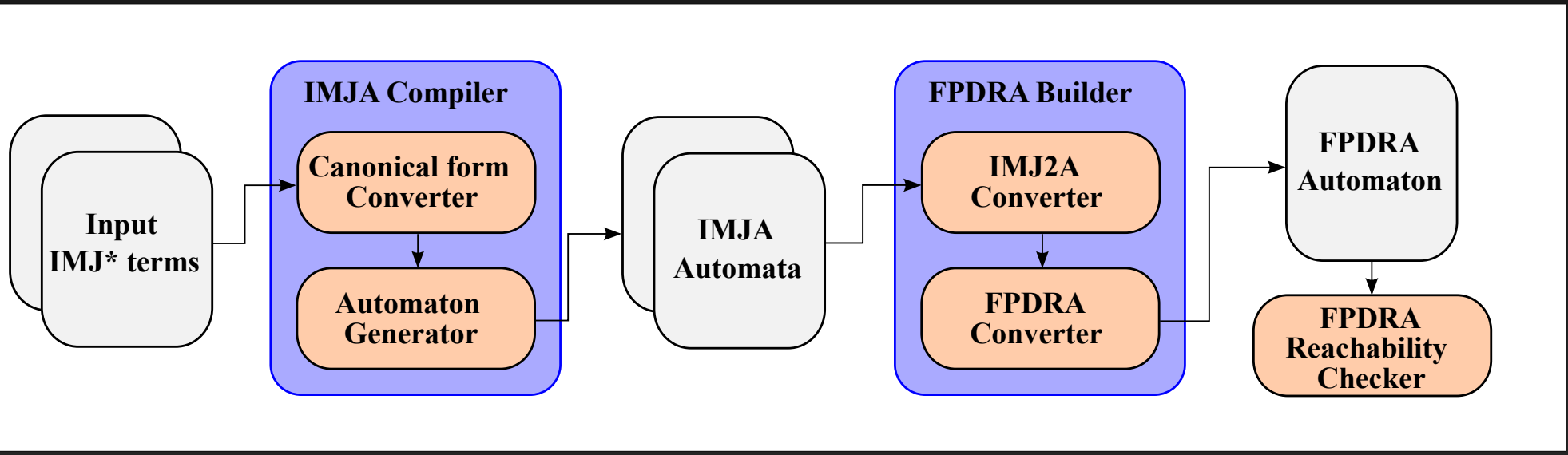
$$M \cong M' \Leftrightarrow [\![M]\!] = [\![M']\!]$$
$$\Leftrightarrow A \sim A' \Leftrightarrow A \otimes A' = 0$$

Implemented in Coneqct!

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 architecture



FPDRA : *Fresh Pushdown Register Automata*

- MRT: Reachability in Pushdown Register Automata, MFCS'14
- MT: Algorithmic Games for Full Ground References, ICALP'12

Coneqct example

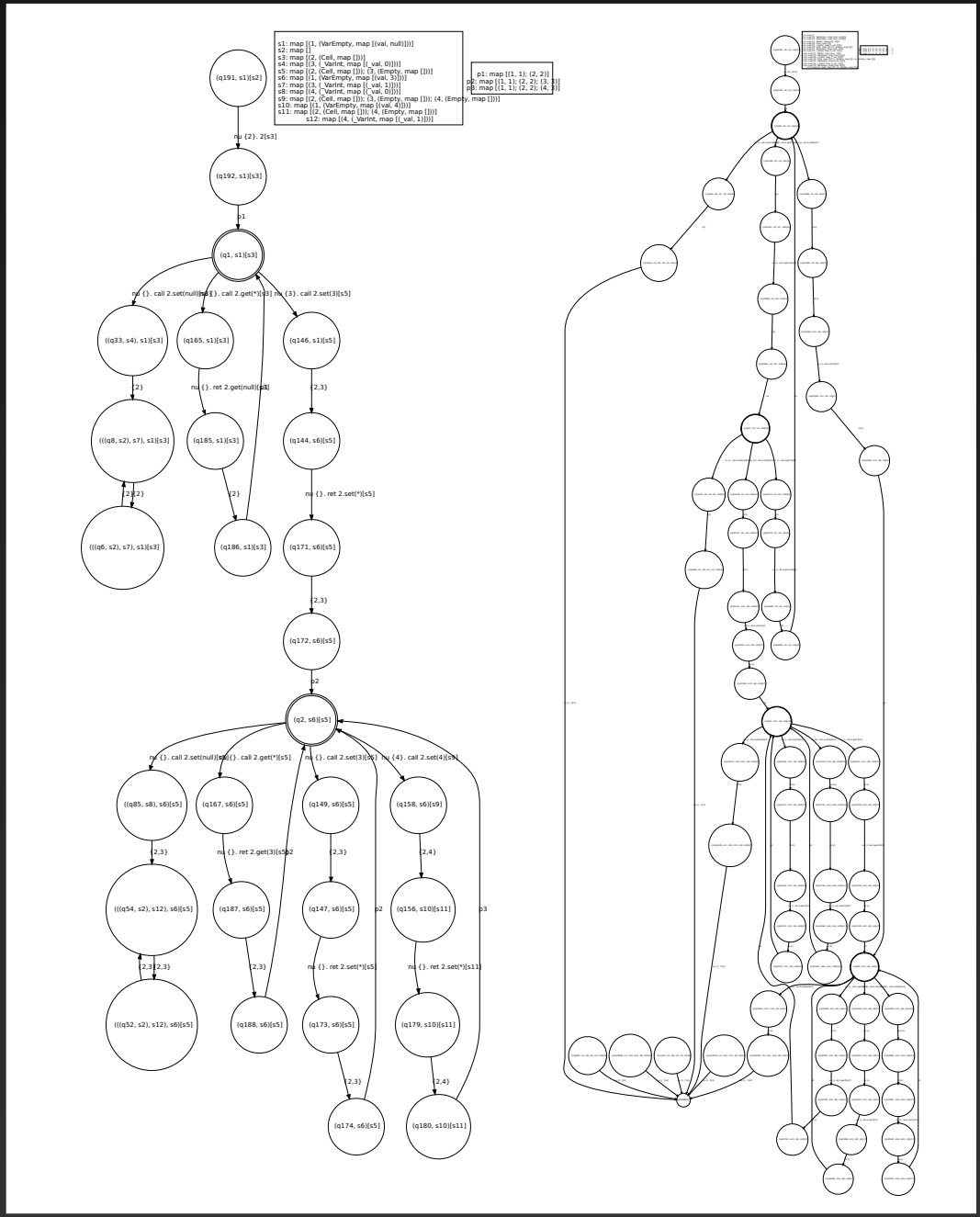
M_1 : let $v = \text{new}\ \{\ \_ : \text{VarEmp}\ \}$ in
new { _: Cell;
get : $\lambda\_.\ v.\text{val}$,
set : $\lambda y.$ if $y = \text{null}$ then div
else $v.\text{val} := y\ ;\ w.\text{val} := y$

M_2 : let $b = \text{new}\ \{\ \_ : \text{VarInt}\ \}$ in
let $v = \text{new}\ \{\ \_ : \text{VarEmp}\ \}$ in
let $w = \text{new}\ \{\ \_ : \text{VarEmp}\ \}$ in
new { _: Cell;
get : $\lambda\_.$ if $b.\text{val} = 1$ then $b.\text{val} := 0\ ;\ v.\text{val}$
else $b.\text{val} := 1\ ;\ w.\text{val}$,
set : $\lambda y.$ if $y = \text{null}$ then div
else $v.\text{val} := y\ ;\ w.\text{val} := y$

Coneqct example

M_1 : let $v = \text{new}\{\_ : \text{VarEmp}\}$ in
new { _: Cell;
get : $\lambda\_.\ v.\text{val}$,
set : $\lambda y.$ if $y = \text{null}$ then div
else $v.\text{val} := y\ ;\ w.\text{val} := y$

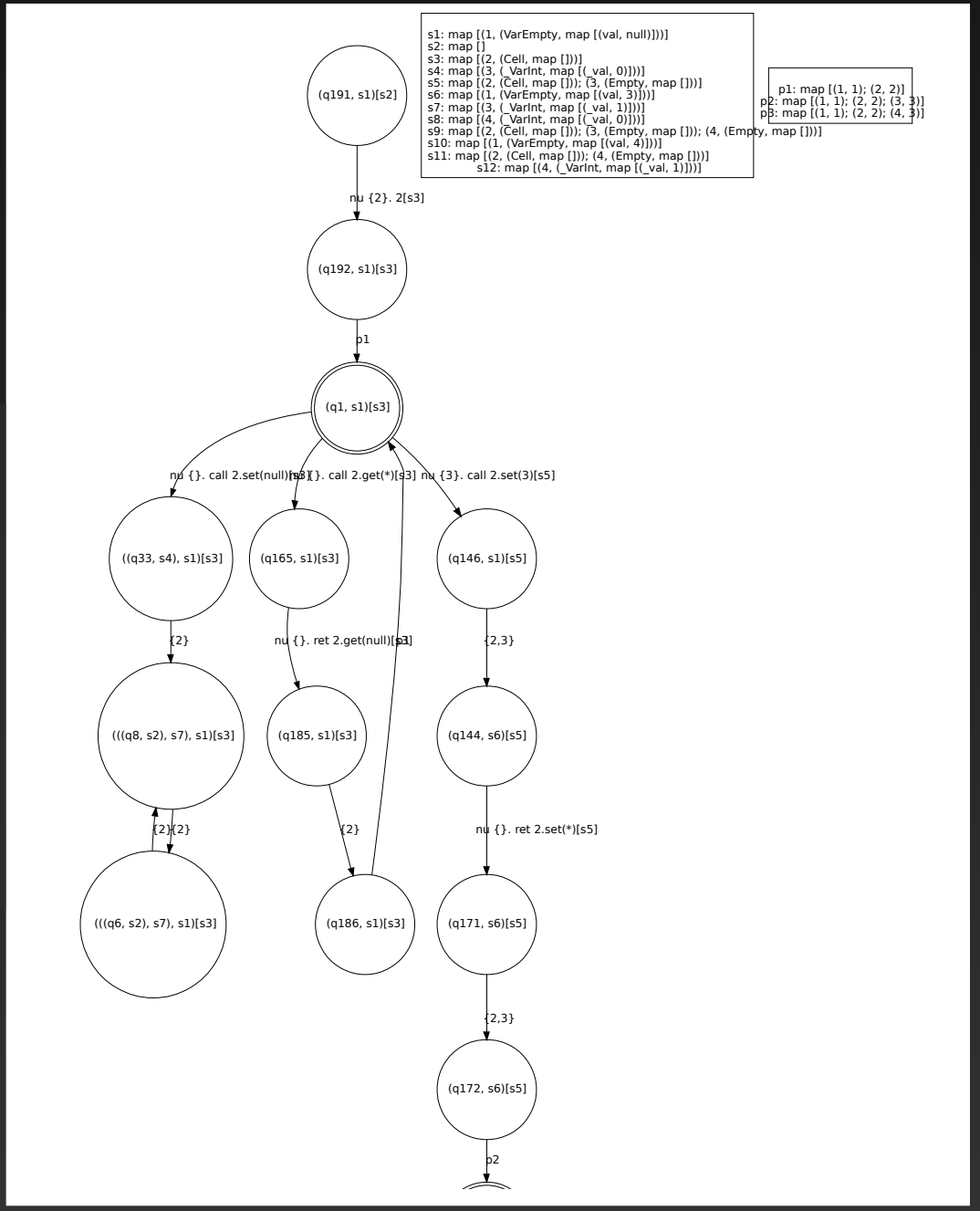
M_2 : let $b = \text{new}\{\_ : \text{VarInt}\}$ in
let $v = \text{new}\{\_ : \text{VarEmp}\}$ in
let $w = \text{new}\{\_ : \text{VarEmp}\}$ in
new { _: Cell;
get : $\lambda\_.$ if $b.\text{val} = 1$ then $b.\text{val} := 0\ ;\ v.\text{val}$
else $b.\text{val} := 1\ ;\ w.\text{val}$,
set : $\lambda y.$ if $y = \text{null}$ then div
else $v.\text{val} := y\ ;\ w.\text{val} := y$



Coneqct example

M_1 : let $v = \text{new}\{\_: \text{VarEmp}\}$ in
new { _: Cell;
get : $\lambda\_.\ v.\text{val}$,
set : $\lambda y.$ if $y = \text{null}$ then div
else $v.\text{val} := y$; $w.\text{val} := y$

M_2 : let $b = \text{new}\{\_: \text{VarInt}\}$ in
let $v = \text{new}\{\_: \text{VarEmp}\}$ in
let $w = \text{new}\{\_: \text{VarEmp}\}$ in
new { _: Cell;
get : $\lambda\_.$ if $b.\text{val} = 1$ then $b.\text{val} := 0$; $v.\text{val}$
else $b.\text{val} := 1$; $w.\text{val}$,
set : $\lambda y.$ if $y = \text{null}$ then div
else $v.\text{val} := y$; $w.\text{val} := y$



Wrapping up

Contextual equivalence

- decidability characterised via games
- automated decision via FPDRAs → Coneqct

Further on

- sound methods for the whole of IMJ
- general model checking and logics

References

- MRT: Game Semantic Analysis of Equivalence in IMJ, ATVA'15
- MRT: A Contextual Equivalence Checker for IMJ*, ATVA'15