## Original citation:

Murawski, Andrzej S., Ramsay, Steven J. and Tzevelekos, Nikos (2015) Game semantic analysis of equivalence in IMJ. In: 13th International Symposium, ATVA 2015,, Shanghai, China, 12-15 Oct 2015. Published in: Automated Technology for Verification and Analysis, 9364 pp. 411-428.

## Permanent WRAP URL:

http://wrap.warwick.ac.uk/78373

## Copyright and reuse:

The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions. Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

## Publisher's statement:

"The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-

## 24953-7 30

## A note on versions:

The version presented here may differ from the published version or, version of record, if you wish to cite this item you are advised to consult the publisher's version. Please see the 'permanent WRAP url' above for details on accessing the published version and note that access may require a subscription.

For more information, please contact the WRAP Team at: wrap@warwick.ac.uk

# Game Semantic Analysis of Equivalence in IMJ 

Andrzej S. Murawski ${ }^{1}$, Steven J. Ramsay ${ }^{1}$, and Nikos Tzevelekos ${ }^{2}$<br>${ }^{1}$ University of Warwick, Coventry, UK $\quad{ }^{2}$ Queen Mary University of London, London, UK


#### Abstract

Using game semantics, we investigate the problem of verifying contextual equivalences in Interface Middleweight Java (IMJ), an imperative object calculus in which program phrases are typed using interfaces. Working in the setting where data types are non-recursive and restricted to finite domains, we identify the frontier between decidability and undecidability by reference to the structure of interfaces present in typing judgments. In particular, we show how to determine the decidability status of problem instances (over a fixed type signature) by examining the position of methods inside the term type and the types of its free identifiers. Our results build upon the recent fully abstract game semantics of IMJ. Decidability is proved by translation into visibly pushdown register automata over infinite alphabets with fresh-input recognition.


## 1 Introduction

Contextual equivalence is the problem of determining whether two (possibly open) program phrases behave equivalently when placed into any possible whole-program context. It is regarded as a gold standard for the identification of behaviours in programming language semantics and is a fundamental concern during refactoring and compiler optimisations. For example, it can be used to determine whether two implementations of an interface behave equivalently irrespective of who might interact with them.

In this work, we undertake an algorithmic study of contextual equivalence for Javastyle objects through the imperative object calculus Interface Middleweight Java (IMJ). IMJ was introduced in [11] as a setting in which to capture the contextual interactions of code written in Middleweight Java [2]. Our aim is to isolate those features of the language, or collections of features taken together, that are so expressive that contextual equivalence becomes undecidable. By such a determination, not only do we gain insight into the power (or complexity) of the features, but also we are able to design complementary fragments for which we have decision procedures. The result of our study is the first classification of decidable cases for contextual equivalence in a core fragment of Java and, on the conceptual front, an exposition of the fundamental limits of automated verification in this setting.

We start delineating the decidable cases by eliminating two features that clearly make IMJ Turing-complete, namely, recursive types and infinite data domains (e.g. unbounded Integers). Hence, our starting point is a finitary restriction of IMJ, in which these two features have been removed. Next we uncover two less obvious features that make termination undecidable (note that termination is a special case of contextual equivalence with skip): storage of method-carrying objects in fields and unrestricted recursion. We show that if either of these resources is available then it is possible to construct a program which simulates a queue machine. In contrast, if the storage of
method-carrying objects is banned and recursion discarded in favour of iteration, we obtain a fragment with decidable termination. Consequently, we need to work with the iterative fragment in which storage of method-carrying objects is prohibited.

Returning to the general case of contextual equivalence, recall that it concerns program phrases which are not necessarily closed or of type void, which leads us to analyse the problem in terms of the result type and the kinds of free variables that may occur in the phrase. When we consider the free variables of a phrase, we find that equivalence is undecidable whenever the phrase relies on a free variable that is an object whose method(s) accept method-carrying objects as parameters, irrespective of the type of the phrase itself. When we consider the type of a program phrase, we find that undecidability is inevitable whenever the phrase:

1. is an object whose method(s) return method-carrying objects, or
2. is an object whose method(s) require a parameter that is itself an object whose method(s) accept method-carrying objects as parameters, irrespective of the free variables upon which the phrase depends.
In contrast, we prove that equivalence is decidable for the class of program phrases that avoid the three criteria. This class is constrained but it still remains a non-trivial object-oriented language: fields cannot store method-carrying objects, but objects with methods can be created at will. Inheritance and encapsulation are supported fully.

Both our undecidability and decidability arguments are enabled by the fully abstract game semantics of IMJ [11], which characterises contextual equivalence of IMJ program phrases by means of strategies (sets of interaction traces which capture the observable behaviour of a program). For undecidability, we observe that, in each of the three cases mentioned above, the patterns of interaction that arise between the phrase and the contexts with which it can be completed are expressive enough to encode the runs of a queue machine. On the other hand, to prove decidability, we show that (in the relevant cases) the corresponding strategies can be related to context-free languages over infinite alphabets. More precisely, we develop a routine which, starting from program phrases, can construct variants of pushdown register automata [13] that represent the associated game semantics. In this way, the problem is ultimately reduced to emptiness testing for this class of automata, which is known to be decidable.

Related Work. We believe we are the first to present a fully automated method for proving contextual equivalences in a Java-like setting, accompanied by a systematic analysis of decidable cases. Contextual equivalence is well known to pose a challenge to automated verification due to quantification over contexts. The quest for obtaining more direct methods of attack on the problem in the Java setting has underpinned a great deal of semantic research, mainly using operational approaches [177|8|6|16], but this did not lead to decision procedures. In our case, the potential for automation stems from the compositionality of the underlying semantics, which allows for a compositional translation of terms into automata in the decidable cases. Previous work based on games-based verification was mainly concerned with various fragments of ML equipped with storage of ground-type values [5]10]. In contrast, in this paper we tackle richer interactions of objects equipped with methods. Compared with these fragments of ML, IMJ contexts are more discriminating, because objects provide a modicum of higher-order state. This motivates our independent study.

$$
\begin{aligned}
& \text { Types } \ni \theta::=\text { void } \mid \text { int } \mid I \quad \text { IDfns } \ni \Theta::=\epsilon|(\mathrm{f}: \theta), \Theta|(\mathrm{m}: \vec{\theta} \rightarrow \theta), \Theta \\
& \text { MImps } \ni \mathcal{M}::=\epsilon \mid(\mathrm{m}: \lambda \vec{x} . M), \mathcal{M} \quad \text { ITbls } \ni \Delta::=\epsilon|(I \equiv \Theta), \Delta|(I\langle I\rangle \equiv \Theta), \Delta \\
& \text { Terms } \ni M::=x \mid \text { null }|a| i \mid \text { if } M \text { then } M \text { else } M|M \oplus M| M ; M|(I) M| M=M \\
& |M . \mathrm{f}| M . \mathrm{m}(\vec{M})|\operatorname{new}\{x: I ; \mathcal{M}\}| M . \mathrm{f}:=M \mid \text { skip } \mid \text { let } x=M \text { in } M \mid \text { while } M \text { do } M \\
& \overline{\Delta \mid \Gamma \vdash x: \theta}(x: \theta) \in \Gamma \quad \overline{\Delta \mid \Gamma \vdash a: I}{ }^{(a: I) \in \Gamma} \quad \overline{\Delta \mid \Gamma \vdash \operatorname{skip}: \text { void }} \quad \overline{\Delta \mid \Gamma \vdash \operatorname{null}: I} I \in \operatorname{dom}(\Delta) \\
& \frac{i \in[0, \text { MAXINT }]}{\Delta \mid \Gamma \vdash i \text { : int }} \quad \frac{\Delta \mid \Gamma \vdash M, M^{\prime}: \text { int }}{\Delta \mid \Gamma \vdash M \oplus M^{\prime}: \text { int }} \quad \frac{\Delta \mid \Gamma \vdash M: \text { int } \quad \Delta \mid \Gamma \vdash M^{\prime}: \text { void }}{\Delta \mid \Gamma \vdash \text { while } M \text { do } M^{\prime}: \text { void }} \\
& \frac{\Delta \mid \Gamma \vdash M: \text { int } \quad \Delta \mid \Gamma \vdash M^{\prime}, M^{\prime \prime}: \theta}{\Delta \mid \Gamma \vdash \text { if } M \text { then } M^{\prime} \text { else } M^{\prime \prime}: \theta} \quad \frac{\bigwedge_{i=1}^{n}\left(\Delta \mid \Gamma \uplus\left\{\vec{x}_{i}: \vec{\theta}_{i}\right\} \vdash M_{i}: \theta_{i}\right)}{\Delta \mid \Gamma \vdash \mathcal{M}:\left\{\mathrm{m}_{i}: \vec{\theta}_{i} \rightarrow \theta_{i} \mid 1 \leq i \leq n\right\}} \\
& \frac{\Delta \mid \Gamma \vdash M, M^{\prime}: I}{\Delta \mid \Gamma \vdash M=M^{\prime}: \mathrm{int}} \quad \frac{\Delta|\Gamma \vdash M: \operatorname{void} \quad \Delta| \Gamma \vdash M^{\prime}: \theta}{\Delta \mid \Gamma \vdash M ; M^{\prime}: \theta} \quad \frac{\Delta|\Gamma \vdash M: I \quad \Delta| \Gamma \vdash M^{\prime}: \theta}{\Delta \mid \Gamma \vdash M . \mathrm{f}:=M^{\prime}: \text { void }} \Delta(I) . \mathrm{f}=\theta \\
& \frac{\Delta \mid \Gamma \vdash M: I}{\Delta \mid \Gamma \vdash M . \mathrm{f}: \theta} \Delta(I) . \mathrm{f}=\theta \quad \frac{\Delta \mid \Gamma \vdash M: I^{\prime}}{\Delta \mid \Gamma \vdash(I) M: I} \underset{\vee I^{\prime} \leq I}{\Delta \vdash I \leq I^{\prime}} \quad \frac{\Delta \mid \Gamma, x: I \vdash \mathcal{M}: \Theta}{\Delta \mid \Gamma \vdash \operatorname{new}\{x: I ; \mathcal{M}\}: I} \Delta(I) \upharpoonright \text { Meths }=\Theta \\
& \frac{\Delta \mid \Gamma \vdash M: I \quad \bigwedge_{i=1}^{n}\left(\Delta \mid \Gamma \vdash M_{i}: \theta_{i}\right)}{\Delta \mid \Gamma \vdash M . \mathrm{m}\left(M_{1}, \cdots, M_{n}\right): \theta} \Delta(I) \cdot \mathrm{m}=\vec{\theta} \rightarrow \theta \quad \frac{\Delta\left|\Gamma \vdash M: \theta^{\prime} \quad \Delta\right| \Gamma, x: \theta^{\prime} \vdash M^{\prime}: \theta}{\Delta \mid \Gamma \vdash \operatorname{let} x=M \operatorname{in} M^{\prime}: \theta}
\end{aligned}
$$

Fig. 1. Definition of $\mathrm{IMJ}_{f}$. Typing rules for terms and method-set implementations.

## 2 Finitary IMJ

We work on Interface Middleweight Java (IMJ), an imperative object calculus based on Middleweight Java [2] which was introduced and examined game semantically in [11]. Here we examine the finitary restriction of IMJ which excludes recursive datatypes and unbounded integers. We call this fragment $\mathrm{IMJ}_{f}$.

We let $\mathbb{A}$ be a countably infinite set of object names, which we range over by $a$ and variants. Names will appear in most syntactic constructs and also in the game model and automata we will consider next. For any construction $X$ that may contain (finitely many) names, we define the support of $X$, denoted $\nu(X)$, to be the set of names occurring in $X$. Moreover, for any permutation $\pi: \mathbb{A} \xlongequal{\leftrightharpoons} \mathbb{A}$, the application of $\pi$ on $X$, written $\pi \cdot X$, to be the structure we obtain from $X$ by transposing all names inside $X$ according to $\pi$. Formally, the above are spelled out in terms of nominal sets [4].

For any pair of natural numbers $i \leq j$ we shall write $[i, j]$ for the set $\{i, i+1, \cdots j\}$. To rule out infinite data domains in $\mathrm{IMJ}_{f}$ we let integers range over [0, MAXInT], where MAXINT is some fixed natural number.
The definition of $\mathrm{IMJ}_{f}$ is given in Figure 1 In more detail, we have the following components:

Intfs, Flds and Meths are sets of interface, field and method identifiers respectively. We range over interfaces by $I$, over fields by f and over methods by m . The types $\theta$ of $\mathrm{IMJ}_{f}$ are selected from Types. An interface definition $\Theta$ is a finite set of typed fields and methods. We require that each identifier $f, m$ can appear at most once in each such definition.

An interface table $\Delta$ is a finite assignment of interface definitions to interface identifiers. We write $I\left\langle I^{\prime}\right\rangle \equiv \Theta$ for interface extension: interface $I$ extends $I^{\prime}$ with fields and methods from $\Theta$. We require that each $I$ can be defined at most once in $\Delta$ (i.e. there is at most one element of $\Delta$ of the form $I: \Theta$ or $\left.I\left\langle I^{\prime}\right\rangle \equiv \Theta\right)$ and if $\left(I\left\langle I^{\prime}\right\rangle \equiv \Theta\right) \in \Delta$ then $\operatorname{dom}\left(\Delta\left(I^{\prime}\right)\right) \cap \operatorname{dom}(\Theta)=\emptyset$. Thus, each $\Theta$ can be seen as a finite partial function $\Theta:($ Flds $\cup$ Meths $) \rightharpoonup$ Types*. We write $\Theta . f$ for $\Theta(f)$, and $\Theta$.m for $\Theta(\mathrm{m})$. Similarly, $\Delta$ defines a partial function $\Delta:$ Intfs $\rightharpoonup I D f n s . I^{\prime} \mathrm{IMJ}_{f}$ there is a recursive types restriction by which recursive (and mutually recursive) definitions of interfaces are not allowed.
$\mathrm{IMJ}_{f}$ terms form the set Terms, where we let $x$ range over a set Vars of variables. Moreover, " $\oplus$ " is selected from some set of binary numeric operations which includes $"="$. Boolean guards are implemented using numbers, with false represented by 0 and true by any other number. $\mathcal{M}$ is a method-set implementation (we stipulate that each m appear at most once in each $\mathcal{M}$ ).
$\mathrm{IMJ}_{f}$ terms are typed in contexts comprising an interface table $\Delta$ and a variable context $\Gamma=\left\{x_{1}: \theta_{1}, \cdots, x_{n}: \theta_{n}\right\} \cup\left\{a_{1}: I_{1}, \cdots, a_{m}: I_{m}\right\}$ such that any interface in $\Gamma$ occurs in dom $(\Delta)$. The typing rules are given in Figure 1 . Here, we write $\Delta(I) \upharpoonright$ Meths to denote the interface definition of $I$ according to $\Delta$ restricted to method specifiers. We write $I \leq I^{\prime}$ to assert that $I$ is a subtype of $I^{\prime}$. The subtyping relation is induced by the use of interface extension in interface definitions as usual. Note, as in Java, downcasting is typable but terms of this form will not make progress in our operational semantics.

In several places in the sequel we will use variable interfaces: for each type $\theta$, we let $\operatorname{Var}_{\theta} \equiv\{$ val: $\theta\}$ be an interface representing a reference of type $\theta$.

Definition 1. We define the sets of term values, heap configurations by:

$$
\text { TVals } \ni v::=\operatorname{skip}|i| \text { null }|a \quad H C n f s \ni V::=\epsilon|(\mathrm{f}: v), V
$$

The set of States $(\ni S)$ is the set of partial functions $\mathbb{A} \rightharpoonup \operatorname{Intfs} \times(H C n f s \times$ MImps $)$. If $S(a)=(I,(V, \mathcal{M})$ ) then we write $S(a): I$, while $S(a)$.f and $S(a)$.m stand for $V . \mathrm{f}$ and $\mathcal{M}$.m respectively, for each $\mathrm{f}, \mathrm{m}$. The transition relation is defined on terms within a state, that is, on pairs $(S, M) \in$ States $\times$ Terms, and is presented in full in [11].

We now define the central problem of our study. Given $\Delta \mid \emptyset \vdash M$ : void, we say that $M$ terminates and write $M \Downarrow$ just if there exists $S$ such that $(\emptyset, M) \rightarrow^{*}(S$, skip $)$.

Definition 2. Given $\Delta \mid \Gamma \vdash M_{i}: \theta(i=1,2)$, we shall say that $\Delta \mid \Gamma \vdash M_{1}: \theta$ contextually approximates $\Delta \mid \Gamma \vdash M_{2}: \theta$ if, for all $\Delta^{\prime} \supseteq \Delta$ and all contexts $C[-]$ such that $\Delta^{\prime} \mid \emptyset \vdash C\left[M_{i}\right]$ : void, if $C\left[M_{1}\right] \Downarrow$ then $C\left[M_{2}\right] \Downarrow$. Two terms are contextually equivalent (written $\Delta \mid \Gamma \vdash M_{1} \cong M_{2}: \theta$ ) if they approximate each other.

Let $\mathcal{X}$ range over subsets of $\mathrm{IMJ}_{f}$. The equivalence problem for $\mathcal{X}$ is to decide equivalence of arbitrary $\mathcal{X}$-terms (under general $\mathrm{IMJ}_{f}$ contexts).

$$
\mathcal{X} \text {-EQUIV: Given } \mathcal{X} \text {-terms } \Delta \mid \Gamma \vdash M_{1}, M_{2}: \theta \text {, does } \Delta \mid \Gamma \vdash M_{1} \cong M_{2} \text { hold? }
$$

Example 3 ([/8]). Let $\Delta=\left\{\right.$ Empty, Cell, Var $\left._{E_{m p t y}}, \operatorname{Var}_{\text {int }}\right\}$, where Empty is the empty interface (no fields or methods) and Cell $\equiv\{$ get : void $\rightarrow$ Empty, set : Empty $\rightarrow$ void $\}$, and consider the terms $\Delta \mid \emptyset \vdash M_{1}, M_{2}$ : Cell:

$$
\begin{aligned}
& M_{1} \equiv \text { let } v=\text { new }\left\{\ldots \text { _Var }_{\text {Empty }}\right\} \text { in } \\
& \text { new \{ _: Cell; } \\
& \text { get : } \lambda_{-} . v . v a l, \\
& \text { set }: \lambda y \text {. if } y=\text { null then div else } v . \text { val }:=y\} \\
& M_{2} \equiv \text { let } b=\text { new }\left\{\_ \text {_: Var } \text { int }\right\} \text { in } \\
& \text { let } v=\text { new }\left\{{ }_{-}: \operatorname{Var}_{\text {Empty }}\right\} \text { in let } w=\text { new }\left\{{ }_{-}: \operatorname{Var}_{\text {Empty }}\right\} \text { in } \\
& \text { new \{ _: Cell; } \\
& \text { get : } \lambda_{\text {. }} \text { if } b . v a l=1 \text { then (b.val:=0; v.val) else (b.val:=1; w.val), } \\
& \text { set }: \lambda y \text {. if } y=\text { null then div else } v . v a l:=y ; w . v a l:=y\}
\end{aligned}
$$

Here div stands for while 1 do skip. We saw in [11] that $\Delta \mid \emptyset \vdash M_{1} \cong M_{2}$ : Cell, by comparing the game semantics of $M_{1}$ and $M_{2}$. The equivalence can be verified automatically, as the terms reside in the decidable fragment for equivalence (we revisit these terms in Section 6.

## 3 Preliminary Analysis: Termination

Since termination can be reduced to contextual equivalence, a good starting point for analysing fragments of $\mathrm{IMJ}_{f}$ that have decidable equivalence is to exclude those that have undecidable termination. The restrictions on $\mathrm{IMJ}_{f}$ preclude obvious undecidability arguments based on arithmetic or recursive datatypes such as lists. However, in this section we identify two more subtle causes for undecidability of termination: fields containing objects with methods, and recursion.

Theorem 4. The termination problem for $\mathrm{IMJ}_{f}$ is undecidable. In particular, it is undecidable for terms $\Delta \mid \emptyset \vdash M$ : void where

1. there are no recursive definitions but fields can store objects with methods,
2. no field stores a method-carrying object, but there are recursive method definitions.

Proof (sketch). In both cases it is possible to encode a queue machine. Since there are no recursive types the structure of the queue has to be coded into other language features. For example, in 1, the links are formed by capturing an existing object in a closure which forms part of the definition of a method.

It follows that any fragment of $\mathrm{IMJ}_{f}$ containing unrestricted recursion or allowing for fields to store method-carrying objects necessarily also has an undecidable equivalence problem. Since $\mathrm{IMJ}_{f}$ already provides a more restricted form of recursion in the while construct, a natural question is to next ask whether termination is still undecidable in the fragment in which fields are restricted to only store objects without attached methods and recursion is disallowed in favour of iteration.

Definition 5. The method dependency graph of a term $\Delta \mid \Gamma \vdash M: \theta$ has as nodes pairs $(I, m)$ of interface $I \in \operatorname{dom}(\Delta)$ and method $m \in \Delta(I)$ and an edge from $(I, m)$ to $\left(J, m^{\prime}\right)$ just if there is a subterm of $M$ which has shape new $\left\{x: I ; \mathcal{M}_{1},(m\right.$ : $\left.\left.\lambda \vec{x} . C\left[P . m^{\prime}(\vec{N})\right]\right), \mathcal{M}_{2}\right\}$ with $\Delta \mid \Gamma^{\prime} \vdash P: J^{\prime}$ and $J \leq J^{\prime}$. That is, such an edge exists just if there is an instance of interface $I$ whose implementation of $m$ depends upon $J . m^{\prime}$. We say a term $\Delta \mid \Gamma \vdash M: \theta$ is iterative just if its dependency graph is acyclic.

We shall henceforth consider the fragment of $\mathrm{IMJ}_{f}$ containing iterative terms $\Delta \mid \emptyset \vdash$ $M$ : void in which all fields in $\Delta$ have types conforming to the grammar:

$$
\mathrm{G}::=\operatorname{void}|\operatorname{int}| \overrightarrow{f: G}
$$

where we write $\overrightarrow{f: G}$ to mean an interface identifier that is declared to contain only some number of fields, whose types again conform to G. We call such types ground. For this fragment, termination is decidable.

Theorem 6. If $\Delta \mid \emptyset \vdash M$ : void is an iterative term and fields in $\Delta$ belong to G then $M \Downarrow$ is decidable.

Proof (sketch). We define a suitable notion of visible state (cf. visible heap of [3]) and show that is has bounded depth. Our definition is more general than that in [3], where the heap consists of objects which may be linked to other objects through pointer fields, since in $\mathrm{IMJ}_{f}$ objects are also equipped with method implementations.

Next we attack contextual equivalence for the $\mathrm{IMJ}_{f}$ fragment with decidable termination. That is, terms do not use recursion and fields are of ground type. Our approach utilises the game model of IMJ [11], adapted to finite integers, the main ingredients of which are seen next ${ }_{\square}^{1}$ The model makes it possible to analyse the observable computational steps of a program phrase and its environment, and plays a crucial role in our decidability and undecidability proofs.

## 4 The Game Model

Game semantics models computation as an exchange of moves between two players, representing respectively the program (player $P$ ) and its environment (player $O$ ). A program phrase is interpreted as a strategy in the game determined by its type, and the patterns of interaction between the players are made concrete in the plays of the game. Given an IMJ term $\Delta \mid \Gamma \vdash M: \theta$, its game semantics is a strategy: a set of formal interactions (called plays), each of which consists of a sequence of tokens (called moves-with-store) that capture the computational potential of $M$. Moves, plays and strategies will involve names in their constructions (that is, they shall live within nominal sets [4]).

The moves available for play are very specific and depend upon the typing environment $\Delta \mid \Gamma \vdash \theta$. For each type $\theta$, we set $\operatorname{Val}_{\theta}$ to be the set of semantic values of type $\theta$, given by: $V a l_{\text {void }}=\{\star\}, V a l_{\text {int }}=[0$, MAXInt $]$ and $V a l_{I}=\mathbb{A} \cup\{$ nul $\}$. We write Val for the union of all $\mathrm{Val}_{\theta}$ 's and, for each type sequence $\vec{\theta}=\theta_{1}, \cdots, \theta_{n}$, set $\operatorname{Val}_{\vec{\theta}}=\operatorname{Val}_{\theta_{1}} \times \cdots \times \operatorname{Val}_{\theta_{n}}$. We let a stores $\Sigma$ be finite partial functions $\Sigma: \mathbb{A} \rightharpoonup$ Intfs $\times($ Flds $\rightharpoonup \mathrm{Val})$ (from names to object types and field assignments) satisfying two closure conditions. To spell them out, given $\Sigma$ and $v \in V a l$, we first define judgments $\Sigma \vdash v: \theta$ by the following rules.

$$
\frac{v \in V a l_{\mathrm{void}}}{\Sigma \vdash v: \text { void }} \quad \frac{v \in V a l_{\mathrm{int}}}{\Sigma \vdash v: \mathrm{int}} \quad \frac{\Sigma(v): I \vee v=\mathrm{null}}{\Sigma \vdash v: I}
$$

Now, whenever $\Sigma(a)=(I, \phi)$, we also stipulate the following conditions.

[^0]- $\Delta(I) . \mathrm{f}=\theta^{\prime}$ implies that $\phi(\mathrm{f})$ is defined and $\Sigma \vdash \phi(\mathrm{f}): \theta$, where $\theta \leq \theta^{\prime}$.
$-\phi(\mathrm{f})=v$ implies that $\Delta(I) . \mathrm{f}$ is defined and, if $v \in \mathbb{A}$ then $v \in \operatorname{dom}(\Sigma)$.
We let Sto be the set of all stores. Note that, for every store $\Sigma, \nu(\Sigma) \subseteq \operatorname{dom}(\Sigma)$.
Definition 7. Given a typing environment $\Delta \mid \Gamma \vdash \theta$ with $\Gamma=\left\{x_{1}: \theta_{1}, \cdots, x_{n}: \theta_{n}\right.$, $\left.a_{1}: I_{1}, \cdots, a_{m}: I_{m}\right\}$, its moves are $M_{\llbracket \Delta \mid \Gamma \vdash \theta \rrbracket}=M_{\llbracket \Gamma \rrbracket} \cup M_{\llbracket \theta \rrbracket} \cup$ Calls $\cup$ Retns, where

$$
\begin{aligned}
M_{\llbracket \Gamma \rrbracket} & =\left\{\pi \cdot\left(v_{1}, \ldots, v_{n}, a_{1}, \ldots, a_{m}\right) \mid \vec{v} \in \operatorname{Val}_{\vec{\theta}} \wedge \pi: \mathbb{A} \cong\right. \\
\text { Calls } & =\left\{\operatorname{call} a \cdot \mathrm{~m}(\vec{v}) \mid a \in \mathbb{A} \wedge \vec{v} \in \operatorname{Val}^{*}\right\} \\
\text { Retns } & =\{\operatorname{ret} a . \mathrm{m}(v) \mid a \in \mathbb{A} \wedge v \in \operatorname{Val}\}
\end{aligned}
$$

and $M_{\llbracket \theta \rrbracket}=V a l_{\theta}$. A move-with-store is pair of a move and a store, written $m^{\Sigma}$.
A play is a sequence of moves-with-store that adheres to the following grammar,

$$
\begin{aligned}
& P_{\llbracket \Delta \mid \Gamma \vdash \theta \rrbracket}: \\
& X::=\epsilon\left|m_{\Gamma}^{\Sigma} X\right| m_{\Gamma}^{\Sigma} Y m_{\theta}^{\Sigma} X \quad(\text { Well-Bracketing) } \\
& Y::=\epsilon|Y Y|(\operatorname{call} a \cdot \mathrm{~m}(\vec{v}))^{\Sigma} X \\
&. \mathrm{m}(\vec{v}))^{\Sigma} Y(\operatorname{ret} a \cdot \mathrm{~m}(v))^{\Sigma}
\end{aligned}
$$

where $m_{\Gamma}$ and $m_{\theta}$ range over $M_{\llbracket \Gamma \rrbracket}$ and $M_{\llbracket \theta \rrbracket}$ respectively, and satisfies some additional conditions [11]: Frugality, Well-Classing and Well-Calling. A play is called complete if it is of the form $m_{\Gamma}^{\Sigma} Y m_{\theta}^{\Sigma} Y$. We write $P_{\llbracket \Delta \mid \Gamma \vdash \theta \rrbracket}$ for the set of plays over $\Delta \mid \Gamma \vdash \theta$. The first move-with-store of a play is played by player O , and from there on players alternate. In particular, the set of plays of length 1 is equal to $P_{\Delta \mid \Gamma}^{1}=\left\{m^{\Sigma} \mid m \in\right.$ $\left.M_{\llbracket \Gamma \rrbracket} \wedge \Sigma \in \operatorname{Sto} \wedge \nu(m) \subseteq \operatorname{dom}(\Sigma)\right\}$ and its elements called initial moves-with-store.

A strategy in $\llbracket \Delta \mid \Gamma \vdash \theta \rrbracket$ is an even-prefix-closed set of plays from $P_{\llbracket \Delta \mid \Gamma \vdash \theta \rrbracket}$ satisfying the combinatorial conditions of Determinacy, Equivariance and O-closure (cf. [11]). We write comp $(\sigma)$ for the set of complete plays of a strategy $\sigma$.

For each table $\Delta$, games yield a category where the morphisms are strategies and the objects are representations of typing environments $\Gamma$ and types $\theta$. A term-in-context $\Delta \mid \Gamma \vdash M: \theta$ is translated into a strategy in $\llbracket \Delta \mid \Gamma \vdash M: \theta \rrbracket$ in a compositional manner [11]. We give a flavour of this interpretation in the next example.

Example 8. Consider interfaces Var $_{\text {int }}=\{$ val : int $\}, I=\{r u n:$ void $\rightarrow$ void $\}$ and terms $f: I \vdash M_{i}: I$ given below, where div implements divergence, $f$ is a free variable of type $I$, and assert (condition) stands for if condition then skip else div. The following terms live in the fragment for which equivalence will be shown decidable.

$$
\begin{aligned}
& M_{1} \equiv \text { let } x=\text { new }\left\{\_: \operatorname{Var}_{\text {int }}\right\} \text { in } \\
& \text { new \{ _: } I \text {; } \\
& \text { run : } \lambda_{-} \text {. if } x . v a l=0 \text { then (x.val }:=1 ; \text { f.run (); assert (x.val=2)) } \\
& \text { else (if } x \text {.val }=1 \text { then } x \text {.val }:=2 \text { else div) } \\
& \text { \} } \\
& M_{2} \equiv \text { new }\left\{\text { _: } I ; \text { run: } \lambda_{\text {_. }} \text { div }\right\}
\end{aligned}
$$

The two terms are not equivalent, since they can be distinguished by a context that first calls the run method of $M_{1}$ (thus triggering a call to $f \cdot r u n()$ ) and then calls $M_{1}$ 's run
again from within the run method of $f$. This will engage $M_{1}$ in a terminating interaction, while calling $M_{1}$ 's run. On the other hand, as soon as $M_{2}$ 's run method is called, we obtain divergence. In game semantics, this is witnessed by the (unique) complete play of $\llbracket M_{1} \rrbracket$ : $n_{f}^{\Sigma_{0}} n^{\Sigma}$ call $n$.run $(\star)^{\Sigma}$ call $n_{f} \cdot \operatorname{run}(\star)^{\Sigma}$ call n.run $(\star)^{\Sigma}$ ret n.run $(\star)^{\Sigma}$ ret $n_{f} \cdot \operatorname{run}(\star)^{\Sigma}$ ret $n \cdot \operatorname{run}(\star)^{\Sigma}$, where $\Sigma_{0}=\left\{n_{f}: I\right\}$ and $\Sigma=\Sigma_{0} \cup\{n: I\}$. Note that the moves in the play correspond exactly to those interactions that happen between the term and its environment (the initial moves $n_{f}^{\Sigma_{0}}$ and $n^{\Sigma}$ correspond to the environment presenting the object $n_{f}$ which instantiates the free variable $f$ and the program presenting the object $n$ to which $M_{1}$ evaluates). Computation that is local to $M_{1}$ is not part of the play. On the other hand, no such play exists in $\llbracket M_{2} \rrbracket$.

Theorem 9 ([11]). For all $\mathrm{IMJ}_{f}$ terms $\Delta \mid \Gamma \vdash M_{1}, M_{2}: \theta, M_{1} \cong M_{2}$ if and only if $\operatorname{comp}\left(\llbracket M_{1} \rrbracket\right)=\operatorname{comp}\left(\llbracket M_{2} \rrbracket\right)$.

## 5 Contextual Equivalence is Undecidable

For a start, we identify three undecidable cases. They will inform the design of the decidable fragment in the next section. Each undecidable case is characterised by the presence of a method in a particular place of the typing judgment $\Delta \mid \Gamma \vdash M: \theta$. To establish undecidability, let $\mathcal{Q}=\left\{Q, Q_{E}, Q_{D}, \delta, q_{I}, \delta_{E}, \delta_{D}\right\}$, with the $Q_{E}$ the set of states from which an enqueue transition in $\delta_{E}$ will fire and $Q_{D}$ the set of states from which a dequeue transition in $\delta_{D}$ will fire. We shall construct $\mathrm{IMJ}_{f}$ terms $\Delta \mid \Gamma \vdash M_{1}, M_{2}: \theta$ such that $M_{1} \cong M_{2}$ if and only if $\mathcal{Q}$ does not halt. Neither recursion nor iteration will be used in the argument and all fields will belong to G . The terms $M_{1}, M_{2}$ will not simulate queue machines directly, i.e. via termination and constructing a queue. Instead, we shall study the interaction (game) patterns they engage in and find that their geometry closely resembles the queue discipline.

Case 1. In this case the undecidability argument will rely on the interface table $\Delta=\left\{I_{1}, I_{2}, I_{3}, I_{4}\right\}$, where $I_{1} \equiv$ val : int, $I_{3} \equiv$ step $:$ void $\rightarrow$ void, $I_{2} \equiv t m p: I_{1}$ and $I_{4} \equiv$ run $: I_{3} \rightarrow$ void; and terms $\Delta \mid x: I_{4} \vdash M_{1}, M_{2}:$ void. Note that $I_{4}$ occurs in the context and the argument type of one of its methods contains a method. We give the relevant terms $M_{1}, M_{2}$ below, where $N_{1} \equiv \operatorname{div}$ and $N_{2} \equiv \operatorname{assert}($ global.val $=$ halt $)$.

```
let global = new {_: I I }, aux = new {_:I I } in
aux.tmp := new{_:I I };
x.run( new{ _: I3;
    step: \lambda.. assert (global.val }\in\mp@subsup{Q}{E}{})
                    let mine = new {_: II }, prev = aux.tmp in
                    aux.tmp := mine;
            mine.val := \pi
            x.run( new{ _: I I ;
            step: \lambda_. assert(global.val }\in\mp@subsup{Q}{D}{})
                                    assert (prev.val =0 and mine.val }\not=0\mathrm{ );
                    global.val := \delta (global.val,mine.val);
                    mine.val := 0;
                    if (aux.tmp = mine) then global.val := halt });
                Ni });
Ni
```

The terms $M_{i}$ are constructed in such a way that any interaction with them results in a call to $x . r u n$ (line 3 ). The argument is a new object of type $I_{3}$, i.e. it is equipped with a step method. Calls to that step method are used to mimic each enqueuing: the value is stored in a local variable mine, and global is used to keep track of the state of the machine. Note that a call to step triggers a call to $x$.run whose argument is another new object of type $I_{3}$ (line 8 ) with a different step method, which can subsequently be used to interpret the dequeuing of the stored element once it becomes the top of the queue. The queue discipline is enforced thanks to private variables of type $I_{2}$, which store references to stored elements as they are added to the queue: once a new value is added a pointer to the previous value is recorded in prev (line 5). To make sure that only values at the front of the queue are dequeued we insert assertions that checks if the preceding value was already dequeued (line 10). Other assertions (lines 4, 9) guarantee that we model operations compatible with the state of the machine. Finally, the difference between $N_{1}$ and $N_{2}$ makes it possible to detect a potential terminating run of the queue.

Theorem 10. Contextual equivalence is undecidable for terms of the form $\Delta \mid x: I_{4} \vdash$ $M_{1}, M_{2}$ : void, where $M_{1}, M_{2}$ are recursion- and iteration-free.

Using a similar approach, though with different representation schemes for queue machines, one can show two more cases undecidable. We mention below the interfaces used in each case.

Case 2. $\Delta=\left\{I_{1}, I_{2}, I_{3}, I_{5}\right\}$, where $I_{5} \equiv e n q$ : void $\rightarrow I_{3}$. An analogue argument can then be formulated for terms $\Delta \mid \emptyset \vdash M: I_{5}$. Note that $I_{5}$ is used as a term type and that it features a method whose result type also contains a method.

Case 3. $\Delta=\left\{I_{1}, I_{2}, I_{3}, I_{4}, I_{6}\right\}$, where $I_{6} \equiv e n q: I_{4} \rightarrow$ void. An analogue argument can then be formulated for terms $\Delta \mid \emptyset \vdash M: I_{6}$. Note that $I_{6}$ is used as a term type and that it contains a method whose argument type has a method.

In the next section we devise a fragment of $\mathrm{IMJ}_{f}$ that forbids each of these cases, which leads to decidability of $\cong$.

## 6 Equivalence is Decidable for IMJ*

In this section we delineate a fragment of $\mathrm{IMJ}_{f}$ that circumvents the undecidable cases identified in the previous section. In order to avoid Case 1, in the context we shall only allow interfaces conforming to the grammar given below.

$$
L::=\operatorname{void}|\operatorname{int}|(\overrightarrow{f: G}, \overrightarrow{m: \vec{G} \rightarrow L})
$$

Note that this prevents methods from having argument types containing methods. Put otherwise, $L$ interfaces are first-order types.

Similarly, in order to avoid Cases 2 and 3, we restrict term types to those generated by the grammar R on the left below:

$$
R::=\operatorname{void}|\operatorname{int}|(\overrightarrow{f: G}, \overrightarrow{m: \vec{L} \rightarrow G}) \quad B::=\operatorname{void}|\operatorname{int}|(\overrightarrow{f: G}, \overrightarrow{m: \vec{G} \rightarrow G})
$$

Observe that the intersection of $L$ and $R$ is captured by $B$.
Using the above restrictions, we define $\mathrm{IMJ*}$ to be a fragment of $\mathrm{IMJ}_{f}$ consisting of iterative terms $\Delta \mid \Gamma \vdash M: \theta$ such that $\operatorname{cod}(\Gamma) \subseteq \mathrm{L}$ and $\theta \in \mathrm{R}$. Due to asymmetries between $L$ and $R$, we do not rely on the standard inductive definition of the syntax and
define it by a new grammar, given below. Note that in the grammar below the types of $x, y$ are required to be in L , and the type of $M$ is in R . Sometimes the types of $x, y$ and $M$ have to match (e.g. in let $x=M$ in $M^{\prime}$ ), which enforces them to be in B . We write $x^{\mathrm{B}}$ to say that $x$ must be in B and not only in L .

$$
\begin{aligned}
& M::=\text { null }\left|x^{\mathrm{B}}\right| i \mid \text { skip }|\operatorname{new}\{x: \mathrm{R} ; \overrightarrow{\mathrm{m}: \lambda \vec{x} \cdot M}\}| x=x^{\prime}\left|a^{\mathrm{B}}\right| M ; M^{\prime} \mid M . \mathrm{f} \\
& \mid \text { if } M \text { then } M^{\prime} \text { else } M^{\prime \prime}\left|M . \mathrm{f}:=M^{\prime}\right| M=M^{\prime}|M \cdot \mathrm{~m}(\vec{M})|(I) M \mid M \oplus M^{\prime} \\
& \mid \text { while } M \text { do } M^{\prime} \mid \text { let } x=(I) y \text { in } M|y . \mathrm{f}| \text { let } x=y \cdot \mathrm{~m}(\vec{M}) \text { in } M \mid \text { let } x=M \text { in } M^{\prime}
\end{aligned}
$$

Our approach for deciding equivalence in IMJ* consists in translating terms into automata that precisely capture their game semantics, and solving the corresponding language equivalence problem for the latter. The automata used for this purpose are a special kind of automata over infinite alphabets, defined in the next section.

### 6.1 Automata

The automata we consider operate over an infinite alphabet $\mathbb{W}$ containing moves-withstore:

$$
\mathbb{W}=\left\{m^{\Sigma} \mid m \in M_{\llbracket \Delta \mid \Gamma \vdash \theta \rrbracket} \wedge \Sigma \text { a store } \wedge \nu(m) \subseteq \operatorname{dom}(\Sigma)\right\}
$$

for given $\Gamma, \Delta, \theta$. Each automaton will operate on an infinite fragment of $\mathbb{W}$ with stores of bounded size. Thus, the sequences accepted by our automata correspond to representations of plays where the domain of the store has been restricted to a bounded size. Due to bounded visibility, such a representation will be complete.

Even with bounded stores, $\mathbb{W}$ is infinite due to the presence of elements of $\mathbb{A}$ (the set of object names) in it. In order to capture names in a finite manner, our automata use a register mechanism. That is, they come equipped with a finite number of registers where they can store names, compare them with names read from the input, and update them to new values during their operation. Thus, each automaton transition refers to names symbolically: via the indices of the registers where they can be found. The automata are also equipped with a visible pushdown stack which can also store names; these names are communicated between the registers and the stack via push/pop operations. They are therefore pushdown extensions of Fresh-Register Automata [15]10].

To spell out formal definitions, let R refer to the number of registers of our automata and let $\mathbb{C}_{s t}$ be a finite set of stack symbols. These will vary between different automata. We set:

$$
\mathbb{C}=\{\star, \text { nul }\} \cup[0, \text { MAXINT }] \quad \mathbb{C}_{r}=\left\{\mathbf{r}_{i} \mid i \in[1, \mathrm{R}]\right\}
$$

$\mathbb{C}$ is the set of constant values that can appear in game moves. $\mathbb{C}_{r}$ are the constants whose role is to refer to the registers symbolically (e.g. $r_{2}$ refers to register number 2).

Definition 11. Given some interface table $\Delta$, we let the set of symbolic values be Val $_{S}=\mathbb{C} \cup \mathbb{C}_{r}$. The sets of symbolic moves, stores and labels are given by:

$$
\begin{aligned}
\text { Mov }_{\mathrm{S}} & =\text { Val }_{\mathrm{S}}^{*} \cup\left\{\text { call }_{i} \cdot \mathrm{~m} \cdot(\vec{x}), \text { ret } \mathrm{r}_{i} \cdot \mathrm{~m}(x) \mid x \vec{x} \in \text { Val }_{\mathrm{S}}^{+}\right\} \\
\text {Sto }_{\mathrm{S}} & =\mathbb{C}_{\mathrm{r}} \rightharpoonup \operatorname{Intfs} \times\left(\text { Flds } \rightharpoonup \text { Val }_{\mathrm{S}}\right)
\end{aligned}
$$

and $L a b_{\mathrm{S}}=M o v_{\mathrm{S}} \times S t o_{\mathrm{S}}$. For any $x \in V a l_{\mathrm{S}} \cup \operatorname{Mov}_{\mathrm{S}} \cup S t o_{\mathrm{S}} \cup L a b_{\mathrm{S}}$, we write $\nu_{\mathrm{r}}(x)$ for the set of registers appearing in $x$. We range over symbolic values by $\ell$ and variants,
symbolic moves by $\mu$ etc, and symbolic stores by $S$ etc. For symbolic labels we may use variants of $\Phi$ or, more concretely, $\mu^{S}$.

The semantics of our automata employs assignments of names to registers. Given such an assignment, we can move from symbolic entities to non-symbolic ones.

Definition 12. The set of register assignments is Reg $=\left\{\rho: \mathbb{C}_{r} \cong \mathbb{A}\right\}$ and contains all partial injections from registers to names. Given $\rho \in \operatorname{Reg}$ and any $x \in \operatorname{Val}_{\mathrm{S}} \cup$ $M o v_{\mathrm{S}} \cup S t o_{\mathrm{S}} \cup L a b_{\mathrm{S}}$, we can define the non-symbolic counterpart of $x$ :

$$
\rho(x)=x\left[\rho\left(\mathrm{r}_{1}\right) / \mathrm{r}_{1}\right] \cdots\left[\rho\left(\mathrm{r}_{\mathrm{R}}\right) / \mathrm{r}_{\mathrm{R}}\right]
$$

where substitution is defined by induction on the syntax of $x$. In particular, $\rho(x)$ is undefined if, for some index $i, \mathrm{r}_{i} \in \nu_{\mathrm{r}}(x)$ but $\rho\left(\mathrm{r}_{i}\right)$ is not defined.

The combination of symbolic labels with register assignments allows us to capture elements of $\mathbb{W}$. That is, whenever an automaton is at a state where a transition with label $\mu^{S}$ can be taken and the current register content is $\rho$, the automaton will perform the transition and accept the letter $\rho\left(\mu^{S}\right) \in \mathbb{W}$.

The pushdown stack that we will be using is going to be of the visible kind: stack operations will be stipulated by the specific symbolic label of a transition. We thus assume that the set of symbolic moves be partitioned into three parts ${ }^{2}$ which in turn yields a partition of symbolic labels:

$$
M o v_{\mathrm{S}}=\operatorname{Mov}_{\text {push }} \uplus \operatorname{Mov}_{\text {pop }} \uplus \operatorname{Mov}_{\text {noop }} \quad L a b_{\alpha}=\left\{\mu^{S} \in L a b \mid \mu \in \operatorname{Mov}_{\alpha}\right\}
$$

where $\alpha \in\{$ push, pop, noop $\}$. We let $S t k=\left(\mathbb{C}_{\text {st }} \times \text { Reg }\right)^{*}$ be the set of stacks. We shall range over stacks by $\sigma$, and over elements of a stack $\sigma$ by $(s, \rho)$.

Definition 13. The set of transition labels is $T L=T L_{\text {push }} \uplus T L_{\text {pop }} \uplus T L_{\text {noop }}$ where, for $\alpha \in\{$ push, pop, noop $\}$ :

$$
T L_{\alpha}=\left\{(X, \Phi, \phi) \in \mathcal{P}\left(\mathbb{C}_{r}\right) \times L a b_{\alpha} \times \mathrm{S}_{\alpha} \mid X \subseteq \nu_{\mathrm{r}}(\Phi)\right\}
$$

and $\mathrm{S}_{\text {push }}=\mathbb{C}_{\text {st }} \times \mathcal{P}\left(\mathbb{C}_{\mathrm{r}}\right), \mathrm{S}_{\text {pop }}=\mathbb{C}_{\text {st }} \times \mathcal{P}\left(\mathbb{C}_{\mathrm{r}}\right)^{2}, \mathrm{~S}_{\text {noop }}=\{()\}$.
We range over $T L$ by $\nu X .(\Phi, \phi)$, where if $X=\emptyset$ then we may suppress the $\nu X$ part altogether; if $X$ is some singleton $\left\{\mathrm{r}_{i}\right\}$ then we may shorten $\nu\left\{\mathrm{r}_{i}\right\}$ to $\nu \mathrm{r}_{i}$. On the other hand, $\phi$ can either be:

- a push pair $(s, Z)$, whereby we may denote $\nu X .(\Phi, \phi)$ by $\nu X . \Phi /(s, Z)$;
- a pop triple $(s, Y, Z)$, in which case we may write $\nu X . \Phi,(s, Y, Z)$;
- or a no-op (), and we may simply write $\nu X . \Phi$.

A transition $\nu X .(\Phi, \phi)$ can thus be seen as doing three things $s^{3}$

- it refreshes the names in registers $X$ (i.e. $\nu X$ stands for "new $X$ ");
- it accepts the letter $\rho(\Phi) \in \mathbb{W}$, where $\rho$ is the "refreshed" assignment;
- it performs the stack operations stipulated by $\phi$.

[^1]These actions will be described in detail after the following definition.
Definition 14. Given a number of registers R, an IMJ-automaton is a tuple $\mathcal{A}=$ $\left\langle Q, q_{0}, X_{0}, \delta, F\right\rangle$ where:

- $Q$ is a finite set of states, partitioned into $Q_{O}$ ( $O$-states) and $Q_{P}$ ( $P$-states);
- $q_{0} \in Q_{P}$ is the initial state; $F \subseteq Q_{O}$ are the final ones;
- $X_{0} \subseteq \mathbb{C}_{\mathrm{r}}$ is the set of initially non-empty registers;
$-\delta \subseteq\left(Q_{P} \times T L_{P O} \times Q_{O}\right) \cup\left(Q_{O} \times T L_{O P} \times Q_{P}\right) \cup\left(Q_{P} \times \mathcal{P}\left(\mathbb{C}_{r}\right) \times Q_{P}\right) \cup\left(Q_{O} \times\right.$ $\left.\left(\mathcal{P}\left(\mathbb{C}_{\mathrm{r}}\right) \cup\left(\mathbb{C}_{\mathrm{r}} \xlongequal{\cong} \mathbb{C}_{\mathrm{r}}\right)\right) \times Q_{O}\right)$ is the transition relation;
where $T L_{P O}=T L_{\text {noop }} \cup T L_{\text {push }}, T L_{O P}=T L_{\text {noop }} \cup T L_{\text {pop }}$.
We next explain the semantics of IMJ-automata. Let $\mathcal{A}$ be as above. A configuration of $\mathcal{A}$ consists of a quadruple $(q, \rho, \sigma, H) \in Q \times \operatorname{Reg} \times \operatorname{Stk} \times \mathcal{P}_{\mathrm{fn}}(\mathbb{A})$. By saying that $\mathcal{A}$ is in configuration $(q, \rho, \sigma, H)$ we mean that, currently: the automaton state is $q$; the register assignment is $\rho$; the stack is $\sigma$; and all the names that have been encountered so far are those in $H$ (i.e. $H$ stands for the current history).

Suppose $\mathcal{A}$ is at a configuration $(q, \rho, \sigma, H)$. If $q \xrightarrow{\nu X .\left(\mu^{S}, \phi\right)} q^{\prime}$ then $\mathcal{A}$ will accept an input $m^{\Sigma} \in \mathbb{W}$ and move to state $q^{\prime}$ if the following steps are successful.

- If $\mu^{S} \in L a b_{\text {pop }}$ and $\phi=(s, Y, Z)$ then $\mathcal{A}$ will check whether the stack has the form $\sigma=\left(s, \rho^{\prime}\right):: \sigma^{\prime}$ with $\operatorname{dom}\left(\rho^{\prime}\right)=Y$ and $\rho, \rho^{\prime}$ being the same in $Z$ and complementary outside it, that is: $\operatorname{dom}(\rho) \cap \operatorname{dom}\left(\rho^{\prime}\right)=Z$ and $\rho \cup \rho^{\prime}$ is a valid assignment. If that is the case, it will pop the top of the stack into the registers, that is, set $\sigma=\sigma^{\prime}$ and $\rho=\rho \cup \rho^{\prime}$.
- $\mathcal{A}$ will update the registers in $X$ with fresh names, that is, it will check whether $\operatorname{dom}(\rho) \cap X=\emptyset$ and, if so, it will set $\rho=\rho\left[\mathrm{r}_{i_{1}} \mapsto a_{1}\right] \cdots\left[\mathrm{r}_{i_{m}} \mapsto a_{m}\right]$, where $i_{1}, \cdots, i_{m}$ is an enumeration of $X$ and $a_{1}, \cdots, a_{m}$ are distinct names such that:
- if $q_{1} \in Q_{O}$ then $a_{1}, \cdots, a_{m} \notin \operatorname{cod}(\rho)$ (locally fresh),
- if $q_{1} \in Q_{P}$ then $a_{1}, \cdots, a_{m} \notin H$ (globally fresh).

In the latter case, $\mathcal{A}$ will set $H=H \cup\left\{a_{1}, \cdots, a_{m}\right\}$.

- $\mathcal{A}$ will check whether $m^{\Sigma}=\rho\left(\mu^{S}\right)$.
- If $\mu^{S} \in L a b_{\text {push }}$ and $\phi=(s, Z)$ then $\mathcal{A}$ will perform a push of all registers in $Z$, along with the constant $s$, that is, it will set $\sigma=(s, \rho \upharpoonright Z):: \sigma$.

Let $\rho^{\prime}$ be the resulting assignment after the above steps have been taken, and similarly for $H^{\prime}$. The semantics of the above transition is the configuration step $(q, \rho, \sigma, H) \xrightarrow{m^{\Sigma}}$ $\left(q^{\prime}, \rho^{\prime}, \sigma^{\prime}, H^{\prime}\right)$.

On the other hand, if $q \xrightarrow{X} q^{\prime}$ then $\mathcal{A}$ will apply the 'mask' $X$ on the registers, that is, set $\rho^{\prime}=\rho \upharpoonright X$, and move to $q^{\prime}$ without reading anything, i.e. $(q, \rho, \sigma, H) \xrightarrow{\epsilon}$ $\left(q^{\prime}, \rho \upharpoonright X, \sigma, H\right)$.

Finally, if $q \xrightarrow{\pi} q$ then $\mathcal{A}$ will permute its registers according to $\pi$, i.e. it will perfom $(q, \rho, \sigma, H) \xrightarrow{\epsilon}\left(q^{\prime}, \rho \circ \pi^{-1}, \sigma, H\right)$.

Given an initial assignment $\rho_{0}$ such that $\operatorname{dom}\left(\rho_{0}\right)=X_{0}$ and taking $H_{0}=\operatorname{cod}\left(\rho_{0}\right)$, the language accepted by $\left(\mathcal{A}, \rho_{0}\right)$ is

$$
\mathcal{L}\left(\mathcal{A}, \rho_{0}\right)=\left\{w \in \mathbb{W}^{*} \mid\left(q_{0}, \rho_{0}, \epsilon, H_{0}\right) \xrightarrow{w}(q, \rho, \epsilon, H) \wedge q \in F\right\} .
$$

We say that $\mathcal{A}$ is deterministic if, from any configuration, there is at most one way to accept each input $x \in \mathbb{W}$.

Let us next look at the two terms from Example 3 and give an automaton which captures their semantics.

Example 15. Consider $\Delta \mid \emptyset \vdash M_{1}, M_{2}$ : Cell from Example 3. The game semantics of the two terms consists of plays of the shape $\star^{\emptyset} n^{\Sigma_{0}} G_{0}^{*} L_{1} G_{1}^{*} \cdots L_{k} G_{k}^{*}$, where:

$$
\begin{aligned}
G_{0} & =\operatorname{call} n \cdot \operatorname{get}(\star)^{\Sigma_{0}} \operatorname{ret} n \cdot \operatorname{get}(\text { nul })^{\Sigma_{0}} \quad G_{i}=\operatorname{call} n \cdot \operatorname{get}(\star)^{\Sigma_{i}} \operatorname{ret} n \cdot \operatorname{get}\left(n_{i-1}\right)^{\Sigma_{i}} \quad(i>0) \\
L_{i} & =\operatorname{call} n \cdot \operatorname{set}\left(n_{i}\right)^{\Sigma_{i}} \operatorname{ret} n \cdot \operatorname{set}(\star)^{\Sigma_{i}} \\
\Sigma_{i} & =\{n \mapsto(\text { Cell, } \emptyset)\} \cup\left\{n_{j} \mapsto(\text { Empty }, \emptyset) \mid j \in[1, i]\right\}
\end{aligned}
$$

for any $n, n_{1}, \cdots, n_{i} \in \mathbb{A}$ with $n \neq n_{j}(j \in[1, i])$. We construct the following IMJautomaton with 2 registers and $X_{0}=\emptyset$.


The automaton represents plays as above, starting from their second move ${ }^{4}$ and matching each $\Sigma_{i}$ to a representation $\hat{\Sigma}_{i}=\Sigma_{0} \cup\left\{n_{i} \mapsto(\right.$ Empty, $\left.\emptyset)\right\}$. The first transition corresponds to the move $n^{\Sigma_{0}}$ : the transition label $\nu \mathrm{r}_{1} . \mathrm{r}_{1}^{S_{1}}$ stipulates that the automaton will accept $n^{\Sigma_{0}}$, for some/any fresh name $n$, and store $n$ in its first register (i.e. register $\mathbf{r}_{1}$ ). Observe how the value $n$, stored in $\mathbf{r}_{1}$, is invoked in later transitions. For instance, the transition labelled call $\mathrm{r}_{1} \cdot \operatorname{get}(\star)^{S_{1}}$ will accept the input call $n \cdot \operatorname{get}(\star)^{\Sigma_{0}}$. Note also that there are two kinds of transitions involving register $r_{2}$. Transitions labelled $\nu \mathbf{r}_{2}$.call $\mathrm{r}_{1}$. $\operatorname{set}\left(\mathrm{r}_{2}\right)^{S_{2}}$ set the value of register $\mathrm{r}_{2}$ to some locally fresh value $n^{\prime \prime}$ and accept call $n . \operatorname{set}\left(n^{\prime \prime}\right)^{\Sigma_{0} \cup\left\{n^{\prime \prime} \mapsto(E m p t y, \emptyset)\right\}}$ (note how, in the transition from $q_{F}^{S_{2}}$ to $q_{1}^{\prime}$, the automaton first clears the contents of $r_{2}$ ). On the other hand, the transition call $\mathrm{r}_{1}$. $\operatorname{set}\left(\mathrm{r}_{2}\right)^{S_{2}}$ corresponds to accepting call $n$.set $\left(n^{\prime}\right)^{\Sigma_{0} \cup\left\{n^{\prime} \mapsto(\text { Empty, Ø) }\}\right.}$ and $n^{\prime}$ is the current value of register $r_{2}$ (i.e. no register update takes place in this case).

### 6.2 Automata for IMJ*

The automata of the previous section are expressive enough to capture the semantics of terms in IMJ*, in the following manner. As seen in the previous example, IMJ-automata do not produce the actual plays of the modelled terms but representations thereof. The reason is that the stores in the game semantics accumulate all names that are played, and are therefore unbounded in size, whereas the size of symbolic stores is by definition

[^2]bounded for each automaton. Our machines represent the actual stores by focussing on the part of the store that the term can access in its current environment (cf. bounded visibility). From a representative "play", where stores are this way bounded, we obtain an actual play by extending stores to their full potential and allowing the values of the added names to be solely determined by O .
Definition 16. Let $s=m_{1}^{\Sigma_{1}} \cdots m_{k}^{\Sigma_{k}}$ and $t=m_{1}^{T_{1}} \cdots m_{k}^{T_{k}}$ be a play and a sequence of moves-with-store over $\Delta \mid \Gamma \vdash \theta$ respectively. We call $s$ an extension of $t$ if $T_{i} \subseteq \Sigma_{i}$ $(i \in[1, k])$ and, for any $i \in[1, k / 2]$, if $a \in \operatorname{dom}\left(\Sigma_{2 i}\right) \backslash \operatorname{dom}\left(T_{2 i}\right)$ then $\Sigma_{2 i}(a)=$ $\Sigma_{2 i-1}(a)$. The set of all extensions of $t$ is ext $(t)$.

We can now state our main translation result. Recall that, for each $\Delta, \Gamma$, we write $P_{\Delta \mid \Gamma}^{1}$ for the set of initial moves-with-store in $\llbracket \Delta \mid \Gamma \vdash \theta \rrbracket$. The set of its initial symbolic moves-with-store is the finite set: $(\Delta \mid \Gamma)=\left\{\mu_{0}^{S_{0}} \in L a b_{\mathrm{S}} \mid \exists \rho_{0} . \rho_{0}\left(\mu_{0}^{S_{0}}\right) \in P_{\Delta \mid \Gamma}^{1}\right\}$. We say that a triple $\left(m^{\Sigma}, \Phi, \rho\right) \in P_{\Delta \mid \Gamma}^{1} \times(\Delta \mid \Gamma) \times$ Reg is compatible if $m^{\Sigma}=\rho(\Phi)$ and $\operatorname{cod}(\rho)=\operatorname{dom}(\Sigma)$, and let $P_{\llbracket \Delta \mid \Gamma \vdash \theta \rrbracket}^{m, \Sigma}$ be the set of plays over $\Delta \mid \Gamma \vdash \theta$ starting with $m^{\Sigma}$.

Theorem 17. Let $\Delta \mid \Gamma \vdash M: \theta$ be an $\mathrm{IMJ}^{*}$ term. We can effectively define a family of deterministic IMJ-automata $\ M D=\left\{0 M D_{\Phi}|\Phi \in \ \Delta| \Gamma D\right\}$ such that

$$
\bigcup_{w \in \mathcal{L}\left(0 M \sum_{\Phi}, \rho\right)} \operatorname{ext}\left(m^{\Sigma} w\right)=\operatorname{comp}(\llbracket \Delta \mid \Gamma \vdash M: \theta \rrbracket) \cap P_{\llbracket \Delta \mid \Gamma \vdash \theta \rrbracket}^{m, \Sigma}
$$

for each compatible $\left(m^{\Sigma}, \Phi, \rho\right)$.
The construction encompasses two stages: first, a syntactic translation of terms into canonical forms (of an appropriate kind) is applied; the latter is followed by a construction of IMJ-automata for terms in canonical form. Both steps are compositional, defined by induction on the term syntax. Let us revisit Example 15 to demonstrate the whole construction.

Example 18. Recall terms $\Delta \mid \emptyset \vdash M_{i}$ : Cell $(i=1,2)$ from Examples 315. Applying our translation on $M_{1}$ and removing unreachable states we obtain the following automaton with $\mathrm{R}=3$ and $X_{0}=\emptyset$.


We can observe that the constructed automaton is obfuscated compared to the one we manually constructed in Example 15, which suggests that the translation can be further
optimised: e.g. states $q_{1}^{\prime}, q_{2}^{\prime}$ and $q_{5}^{\prime}$ can be evidently unified $\left(q_{3}^{\prime}, q_{4}^{\prime}\right.$ too). There is also a symmetry between $q_{F}^{S_{2}}$ and $q_{F}^{S_{3}}$.

Let now $\Delta \mid \Gamma \vdash M_{1}, M_{2}: \theta$ be IMJ* terms. The previous theorem provides us with deterministic IMJ-automata $\mathcal{A}_{i}=\left(M_{i}\right\rangle_{\Phi}(i=1,2)$ representing the complete plays of $\llbracket M_{i} \rrbracket$ which start with $m^{\Sigma}$, for each compatible triple $\left(m^{\Sigma}, \Phi, \rho\right)$. Thus, since the game model is fully abstract with respect to complete plays (Theorem 9), to decide whether $M_{1} \cong M_{2}$ it suffices to check whether $\mathcal{A}_{1}$ and $\mathcal{A}_{2}$ represent the same sets of complete plays, for all compatible ( $m^{\Sigma}, \Phi, \rho$ ).

We achieve the latter by constructing a product-like IMJ-automaton $\mathcal{B}$ which jointly simulates the operation of $\mathcal{A}_{1}$ and $\mathcal{A}_{2}$, looking for possible discordances in their operation which would signal that there is a complete play which one of them can represent but the other cannot. That is, $\mathcal{B}$ operates in joint simulation mode or in divergence mode. When in simulation mode, at each configuration and input move-with-store $\mathrm{m}^{\prime \Sigma^{\prime}}$ :

- if $\mathcal{A}_{1}$ and $\mathcal{A}_{2}$ both accept $m^{\prime \Sigma^{\prime}}$ (modulo extensions) then $\mathcal{B}$ accepts $m^{\prime \Sigma^{\prime}}$ and proceeds its joint simulation of $\mathcal{A}_{1}, \mathcal{A}_{2}$;
- if, say, $\mathcal{A}_{1}$ accepts $m^{\prime \Sigma^{\prime}}$ but $\mathcal{A}_{2}$ cannot accept it, then $\mathcal{B}$ enters divergence mode: it proceeds with simulating only $\mathcal{A}_{1}$ with the target of reaching a final state (dually if $\mathcal{A}_{2} / \mathcal{A}_{1}$ accepts/not-accepts $\left.m^{\Sigma^{\prime}}\right)$. If the latter is successful, then $\mathcal{B}$ will have found a complete play that can be represented by $\mathcal{A}_{1}$ but not by $\mathcal{A}_{2}$.

Because our automata use visibly pushdown stacks and rely on the same partitioning of tags, we can synchronise them using a single stack. In addition, $\mathcal{B}$ needs to keep in its registers the union of the names stored by $\mathcal{A}_{1}$ and $\mathcal{A}_{2}$. Inside its states, $\mathcal{B}$ keeps information on: the current states of $\mathcal{A}_{1}$ and $\mathcal{A}_{2}$; the way the names of its registers correspond to the names in the registers of $\mathcal{A}_{1}$ and $\mathcal{A}_{2}$; the current joint symbolic state (this is for resolving expansions). On the other hand, once in divergence mode, say for $\mathcal{A}_{1}, \mathcal{B}$ operates precisely like $\mathcal{A}_{1}$. The automaton can only accept in divergence mode, if the simulated automaton (here $\mathcal{A}_{1}$ ) accepts.
Theorem 19. For $\Delta \mid \Gamma \vdash M_{1}, M_{2}: \theta$ and $\left(m^{\Sigma}, \Phi, \rho\right)$ as above, we can effectively construct an IMJ-automaton $\mathcal{B}$ such that $\mathcal{L}(\mathcal{B}, \rho)=\emptyset$ iff $\operatorname{comp}\left(\llbracket \Delta \mid \Gamma \vdash M_{1}: \theta \rrbracket\right)=$ $\operatorname{comp}\left(\llbracket \Delta \mid \Gamma \vdash M_{2}: \theta \rrbracket\right)$.

As variants of pushdown fresh-register automata, IMJ-automata have decidable emptiness problem [10]. Moreover, the number of compatible triples $\left(m^{\Sigma}, \Phi, \rho\right)$ is bounded with respect to $\Gamma, \Delta$ modulo name-permutations. This yields the following.
Corollary 20. IMJ*-EQUIV is decidable.

## 7 Conclusion and Further Work

In Section 3 we showed that the ability to construct terms using unrestricted recursion leads to undecidable termination. Hence, we subsequently dropped unrestricted recursion in favour of the natural alternative: iteration through $\mathrm{IMJ}_{f}$ 's while construct. We conclude by discussing a finer gradation of recursion which allows us to study the algorithmic properties of a larger, if perhaps less natural, class of terms: those using only first-order recursion. We show that our bounded-depth visible store argument extends to this new fragment.

Definition 21. We say that a method I.m declared in $\Delta$ is first-order just if $I . m$ has type of shape $(G, \ldots, G) \rightarrow G$. Otherwise we shall say that it is higher-order. Fix a term $\Delta \mid \emptyset \vdash P$ : void. We say that $P$ is 1-recursive just if, whenever there is a cycle $\left(I_{1}, m_{1}\right), \ldots,\left(I_{k}, m_{k}\right)$ in its method dependency graph, then every $I_{i} . m_{i}$ is first-order.

For 1-recursive terms, new objects may be created at every frame of an increasingly large stack of recursive calls and then returned back down the chain so as to be visible in all contexts. However, the number of frames in this call stack which are associated with methods that can pass method-carrying objects as parameters is ultimately bounded by the 1-recursion restriction.

Lemma 22. Let $\Delta$ have only G -valued fields and $\Delta \mid \emptyset \vdash M$ : void be 1-recursive. Then $M$ has bounded-depth visible state.

For the equivalence problem, it is unclear whether our argument carries over to the first-order recursion setting. Recall that we rely on an equivalence-like testing procedure for visibly pushdown register automata. With recursion we cannot hope to remain in the visible setting [9] and the decidability status of language equivalence for general pushdown register automata over infinite alphabets is currently unknown (it would require extending the celebrated result of Sénizergues [14]).

## References

1. E. Ábraham, M. M. Bonsangue, F. S. de Boer, A. Gruener, and M. Steffen. Observability, connectivity, and replay in a sequential calculus of classes. In FMCO, LNCS vol. 3657. 2004.
2. G. Bierman, M. Parkinson, and A. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, Computer Laboratory, University of Cambridge, 2002.
3. A. Bouajjani, S. Fratani, and S. Qadeer. Context-bounded analysis of multithreaded programs with dynamic linked structures. In CAV, pp. 207-220, 2007.
4. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. Formal Aspects of Computing, 13:341-363, 2002.
5. D. Hopkins, A. S. Murawski, and C.-H. L. Ong. A fragment of ML decidable by visibly pushdown automata. In ICALP, LNCS vol. 6756, pp. 149-161. Springer, 2011.
6. R. Jagadeesan, C. Pitcher, and J. Riely. Open bisimulation for aspects. In AOSD. ACM, 2007.
7. A. Jeffrey and J. Rathke. Java Jr: Fully abstract trace semantics for a core Java language. In ESOP, LNCS vol. 3444, pp. 423-438. 2003.
8. V. Koutavas and M. Wand. Reasoning about class behavior. In FOOL/WOOD. 2007.
9. A. S. Murawski, C.-H. L. Ong, and I. Walukiewicz. Idealized Algol with ground recursion and DPDA equivalence. In ICALP, LNCS vol. 3580, pp. 917-929. Springer, 2005.
10. A. S. Murawski and N. Tzevelekos. Algorithmic games for full ground references. In ICALP, LNCS vol. 7392, pp. 312-324. Springer, 2012.
11. A. S. Murawski and N. Tzevelekos. Game semantics for Interface Middleweight Java. In POPL, pp. 517-528. ACM, 2014.
12. J. Rot, F. S. de Boer, and M. M. Bonsangue. Unbounded allocation in bounded heaps. In FSEN, pp. 1-16, 2013.
13. L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In CSL, LNCS, pp. 41-57. Springer, 2006.
14. G. Sénizergues. $L(A)=L(B)$ ? decidability results from complete formal systems. Theoretical Computer Science, 251(1-2):1-166, 2001.
15. N. Tzevelekos. Fresh-register automata. In POPL, pp. 295-306. 2011.
16. Y. Welsch and A. Poetzsch-Heffter. Full abstraction at package boundaries of object-oriented languages. In SBMF, pp. 28-43. Springer, 2011.

[^0]:    ${ }^{1}$ Since the space we can devote in this paper to the exposition of the game model is limited, we kindly refer the reader to [11] for a thorough account.

[^1]:    ${ }^{2}$ The partitioning depends on the type of the move (e.g. only P-calls can be pushes, and only O-returns can be pops) and is made explicit in the automata construction.
    ${ }^{3}$ As we see next, these actions do not necessarily happen in this order (pops happen first).

[^2]:    ${ }^{4}$ This is a technical convenience of the interpretation: as we see next, we translate each canonical term into a family of automata, one per (symbolic) initial move-with-store (here, the unique initial move is $\star^{\emptyset}$ ). Initial states take the initial move as given and are therefore P -states.

