Game Semantic Analysis of Equivalence in IMJ

Andrzej S. Murawski[1], Steven J. Ramsay[1], and Nikos Tzevelekos[2]

[1] University of Warwick, Coventry, UK [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 Java-style 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 its 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 a variant 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 [1,7,8,6,16], but this did not lead to decision procedures. In our case, the potential for automation stems from the compositionality of the underpinning 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.

$$\begin{array}{rl}
\mathit{Types} \ni \theta ::= & \mathsf{void} \mid \mathsf{int} \mid I \qquad \mathit{IDfns} \ni \Theta ::= \epsilon \mid (\mathsf{f}:\theta), \Theta \mid (\mathsf{m}:\vec{\theta} \to \theta), \Theta \\
\mathit{MImps} \ni \mathcal{M} ::= & \epsilon \mid (\mathsf{m} : \lambda \vec{x}.M), \mathcal{M} \qquad \mathit{ITbls} \ni \Delta ::= \epsilon \mid (I \equiv \Theta), \Delta \mid (I\langle I \rangle \equiv \Theta), \Delta \\
\mathit{Terms} \ni M ::= & x \mid \mathsf{null} \mid a \mid i \mid \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M \mid M \oplus M \mid M; M \mid (I)M \mid M = M \\
& \mid M.\mathsf{f} \mid M.\mathsf{m}(\overrightarrow{M}) \mid \mathsf{new}\{\, x : I; \mathcal{M} \,\} \mid M.\mathsf{f} := M \mid \mathsf{skip} \mid \mathsf{let}\, x = M\, \mathsf{in}\, M \mid \mathsf{while}\, M\, \mathsf{do}\, M
\end{array}$$

$$\frac{}{\Delta|\Gamma \vdash x : \theta}\ {\scriptstyle (x:\theta)\in\Gamma} \qquad \frac{}{\Delta|\Gamma \vdash a : I}\ {\scriptstyle (a:I)\in\Gamma} \qquad \frac{}{\Delta|\Gamma \vdash \mathsf{skip} : \mathsf{void}} \qquad \frac{}{\Delta|\Gamma \vdash \mathsf{null} : I}\ {\scriptstyle I\in\mathsf{dom}(\Delta)}$$

$$\frac{i \in [0, \textsc{MaxInt}]}{\Delta|\Gamma \vdash i : \mathsf{int}} \qquad \frac{\Delta|\Gamma \vdash M, M' : \mathsf{int}}{\Delta|\Gamma \vdash M \oplus M' : \mathsf{int}} \qquad \frac{\Delta|\Gamma \vdash M : \mathsf{int} \quad \Delta|\Gamma \vdash M' : \mathsf{void}}{\Delta|\Gamma \vdash \mathsf{while}\, M\, \mathsf{do}\, M' : \mathsf{void}}$$

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

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

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

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

Fig. 1. Definition of 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 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 $\pi : \mathbb{A} \xrightarrow{\cong} \mathbb{A}$, the ***application*** of π on X, written $\pi \cdot X$, to be the structure we obtain from X by transposing all names inside X according to π. Formally, the above are spelled out in terms of *nominal sets* [4].

For any pair of natural numbers $i \le j$ we shall write $[i, j]$ for the set $\{i, i+1, \cdots j\}$. To rule out infinite data domains in IMJ_f we let integers range over $[0, \textsc{MaxInt}]$, where MAXINT is some fixed natural number.

The definition of 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 θ of IMJ_f are selected from Types. An ***interface definition*** Θ 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*** Δ is a finite assignment of interface definitions to interface identifiers. We write $I\langle I'\rangle \equiv \Theta$ for interface extension: interface I extends I' with fields and methods from Θ. We require that each I can be defined at most once in Δ (i.e. there is at most one element of Δ of the form $I : \Theta$ or $I\langle I'\rangle \equiv \Theta$) and if $(I\langle I'\rangle \equiv \Theta) \in \Delta$ then $\mathsf{dom}(\Delta(I')) \cap \mathsf{dom}(\Theta) = \emptyset$. Thus, each Θ can be seen as a finite partial function $\Theta : (\mathit{Flds} \cup \mathit{Meths}) \rightharpoonup \mathit{Types}^*$. We write $\Theta.\mathsf{f}$ for $\Theta(\mathsf{f})$, and $\Theta.\mathsf{m}$ for $\Theta(\mathsf{m})$. Similarly, Δ defines a partial function $\Delta : \mathit{Intfs} \rightharpoonup \mathit{IDfns}$. In IMJ_f there is a ***recursive types restriction*** by which recursive (and mutually recursive) definitions of interfaces are not allowed.

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 $\mathcal{M}$).

IMJ_f terms are typed in contexts comprising an interface table Δ and a variable context $\Gamma = \{x_1 : \theta_1, \cdots, x_n : \theta_n\} \cup \{a_1 : I_1, \cdots, a_m : I_m\}$ such that any interface in Γ occurs in $\mathsf{dom}(\Delta)$. The typing rules are given in Figure 1. Here, we write $\Delta(I) \restriction \mathit{Meths}$ to denote the interface definition of I according to Δ restricted to method specifiers. We write $I \leq I'$ to assert that I is a subtype of I'. 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 θ, we let $\mathsf{Var}_\theta \equiv \{\mathit{val} : \theta\}$ be an interface representing a reference of type θ.

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

$$\mathit{TVals} \ni v ::= \mathsf{skip} \mid i \mid \mathsf{null} \mid a \qquad \mathit{HCnfs} \ni V ::= \epsilon \mid (\mathsf{f} : v), V$$

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

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

Definition 2. Given $\Delta|\Gamma \vdash M_i : \theta$ $(i = 1, 2)$, we shall say that $\Delta|\Gamma \vdash M_1 : \theta$ ***contextually approximates*** $\Delta|\Gamma \vdash M_2 : \theta$ if, for all $\Delta' \supseteq \Delta$ and all contexts $C[-]$ such that $\Delta'|\emptyset \vdash C[M_i] : \mathsf{void}$, if $C[M_1] \Downarrow$ then $C[M_2] \Downarrow$. Two terms are ***contextually equivalent*** (written $\Delta|\Gamma \vdash M_1 \cong M_2 : \theta$) if they approximate each other.

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

$\mathcal{X}$–EQUIV: Given $\mathcal{X}$-terms $\Delta|\Gamma \vdash M_1, M_2 : \theta$, does $\Delta|\Gamma \vdash M_1 \cong M_2$ hold?

Example 3 ([8]). Let $\Delta = \{\mathsf{Empty}, \mathsf{Cell}, \mathsf{Var}_{\mathsf{Empty}}, \mathsf{Var}_{\mathsf{int}}\}$, where Empty is the empty interface (no fields or methods) and $\mathsf{Cell} \equiv \{\mathsf{get} : \mathsf{void} \to \mathsf{Empty},\ \mathsf{set} : \mathsf{Empty} \to \mathsf{void}\}$, and consider the terms $\Delta|\emptyset \vdash M_1, M_2 : \mathsf{Cell}$:

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

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

Here div stands for while 1 do skip. We saw in [11] that $\Delta|\emptyset \vdash M_1 \cong M_2 : \mathsf{Cell}$, 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 IMJ_f that have decidable equivalence is to exclude those that have undecidable termination. The restrictions on 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* IMJ_f *is undecidable. In particular, it is undecidable for terms* $\Delta|\emptyset \vdash M : \mathsf{void}$ *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 IMJ_f containing unrestricted recursion or allowing for fields to store method-carrying objects necessarily also has an undecidable equivalence problem. Since 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 $\Delta|\Gamma \vdash M : \theta$ has as nodes pairs $(I,\ m)$ of interface $I \in \mathsf{dom}(\Delta)$ and method $m \in \Delta(I)$ and an edge from $(I,\ m)$ to $(J,\ m')$ just if there is a subterm of M which has shape $\mathsf{new}\{x\ :\ I;\ \mathcal{M}_1,\ (m\ :\ \lambda\vec{x}.\, C[P.m'(\vec{N})]),\ \mathcal{M}_2\}$ with $\Delta|\Gamma' \vdash P : J'$ and $J \leq J'$. That is, such an edge exists just if there is an instance of interface I whose implementation of m depends upon $J.m'$. We say a term $\Delta|\Gamma \vdash M : \theta$ is ***iterative*** just if its dependency graph is acyclic.

We shall henceforth consider the fragment of IMJ_f containing iterative terms $\Delta|\emptyset \vdash M : \mathsf{void}$ in which all fields in Δ have types conforming to the grammar:

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

where we write $\overrightarrow{f : \mathsf{G}}$ 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 $\Delta|\emptyset \vdash M : \mathsf{void}$ is an iterative term and fields in Δ 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 IMJ_f objects are also equipped with method implementations. □

Next we attack contextual equivalence for the 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.[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 $\Delta|\Gamma \vdash M : \theta$, 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 $\Delta|\Gamma \vdash \theta$. For each type θ, we set Val_θ to be the set of *semantic values* of type θ, given by: $Val_{\mathsf{void}} = \{\star\}$, $Val_{\mathsf{int}} = [0, \textsc{MaxInt}]$ and $Val_I = \mathbb{A} \cup \{\mathsf{nul}\}$. We write Val for the union of all Val_θ's and, for each type sequence $\vec{\theta} = \theta_1, \cdots, \theta_n$, set $Val_{\vec{\theta}} = Val_{\theta_1} \times \cdots \times Val_{\theta_n}$. We let a ***stores*** Σ be finite partial functions $\Sigma : \mathbb{A} \rightharpoonup Intfs \times (Flds \rightharpoonup Val)$ (from names to object types and field assignments) satisfying two closure conditions. To spell them out, given Σ and $v \in Val$, we first define judgments $\Sigma \vdash v : \theta$ by the following rules.

$$\frac{v \in Val_{\mathsf{void}}}{\Sigma \vdash v : \mathsf{void}} \qquad \frac{v \in Val_{\mathsf{int}}}{\Sigma \vdash v : \mathsf{int}} \qquad \frac{\Sigma(v) : I \vee v = \mathsf{null}}{\Sigma \vdash v : I}$$

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

[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.

– $\Delta(I).\mathsf{f} = \theta'$ implies that $\phi(\mathsf{f})$ is defined and $\Sigma \vdash \phi(\mathsf{f}) : \theta$, where $\theta \leq \theta'$.
– $\phi(\mathsf{f}) = v$ implies that $\Delta(I).\mathsf{f}$ is defined and, if $v \in \mathbb{A}$ then $v \in \mathsf{dom}(\Sigma)$.

We let Sto be the set of all stores. Note that, for every store Σ, $\nu(\Sigma) \subseteq \mathsf{dom}(\Sigma)$.

Definition 7. Given a typing environment $\Delta|\Gamma \vdash \theta$ with $\Gamma = \{x_1 : \theta_1, \cdots, x_n : \theta_n, a_1 : I_1, \cdots, a_m : I_m\}$, its ***moves*** are $M_{[\![\Delta|\Gamma\vdash\theta]\!]} = M_{[\![\Gamma]\!]} \cup M_{[\![\theta]\!]} \cup Calls \cup Retns$, where

$$\begin{array}{rcl} M_{[\![\Gamma]\!]} & = & \{\pi \cdot (v_1, ..., v_n, a_1, ..., a_m) \mid \vec{v} \in Val_{\vec{\theta}} \wedge \pi : \mathbb{A} \stackrel{\cong}{\to} \mathbb{A}\} \\ Calls & = & \{\mathsf{call}\ a.\mathsf{m}(\vec{v}) \mid a \in \mathbb{A} \wedge \vec{v} \in Val^*\} \\ Retns & = & \{\mathsf{ret}\ a.\mathsf{m}(v) \mid a \in \mathbb{A} \wedge v \in Val\,\} \end{array}$$

and $M_{[\![\theta]\!]} = Val_\theta$. A ***move-with-store*** for is pair of a move and a store, written m^Σ.

A ***play*** is a sequence of moves-with-store that adheres to the following grammar,

$$\begin{array}{rcll} P_{[\![\Delta|\Gamma\vdash\theta]\!]} & ::= & \epsilon \mid m_\Gamma^\Sigma\, X \mid m_\Gamma^\Sigma\, Y m_\theta^\Sigma\, X & \textit{(Well-Bracketing)} \\ X & ::= & Y \mid Y\,(\mathsf{call}\ a.\mathsf{m}(\vec{v}))^\Sigma\, X & \\ Y & ::= & \epsilon \mid YY \mid (\mathsf{call}\ a.\mathsf{m}(\vec{v}))^\Sigma\, Y\,(\mathsf{ret}\ a.\mathsf{m}(v))^\Sigma & \end{array}$$

where m_Γ and m_θ range over $M_{[\![\Gamma]\!]}$ and $M_{[\![\theta]\!]}$ 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 $P_{[\![\Delta|\Gamma\vdash\theta]\!]}$ for the set of plays over $\Delta|\Gamma \vdash \theta$. 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 $P^1_{\Delta|\Gamma} = \{m^\Sigma \mid m \in M_{[\![\Gamma]\!]} \wedge \Sigma \in Sto \wedge \nu(m) \subseteq \mathsf{dom}(\Sigma)\}$ and its elements called *initial moves-with-store*.

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

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

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

$M_1 \equiv$ let x = new { _: $\mathsf{Var}_{\mathsf{int}}$} in
  new { _: I;
    run : λ_. if $x.val$=0 then ($x.val$:=1; $f.run$(); assert ($x.val$=2))
      else (if $x.val$=1 then $x.val$:=2 else div)
  }
$M_2 \equiv$ new { _: I; *run*: λ_. div }

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 $f.run()$) 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 $[\![M_1]\!]$: $n_f^{\Sigma_0}\, n^{\Sigma}\ \mathsf{call}\, n.run(\star)^{\Sigma}\ \mathsf{call}\, n_f.run(\star)^{\Sigma}\ \mathsf{call}\, n.run(\star)^{\Sigma}\ \mathsf{ret}\, n.run(\star)^{\Sigma}\ \mathsf{ret}\, n_f.run(\star)^{\Sigma}\ \mathsf{ret}\, n.run(\star)^{\Sigma}$, where $\Sigma_0 = \{n_f : I\}$ 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^{Σ} 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 $[\![M_2]\!]$.

Theorem 9 ([11]). *For all* IMJ_f *terms* $\Delta|\Gamma \vdash M_1, M_2 : \theta$, $M_1 \cong M_2$ *if and only if* $\mathsf{comp}([\![M_1]\!]) = \mathsf{comp}([\![M_2]\!])$.

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 characterized by the presence of a method in a particular place of the typing judgment $\Delta|\Gamma \vdash M : \theta$. To establish undecidability, let $\mathcal{Q} = \{Q, Q_E, Q_D, \delta, q_I, \delta_E, \delta_D\}$, with the Q_E the set of states from which an enqueue transition in δ_E will fire and Q_D the set of states from which a dequeue transition in δ_D will fire. We shall construct IMJ_f terms $\Delta|\Gamma \vdash M_1, M_2 : \theta$ 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 $\Delta = \{I_1, I_2, I_3, I_4\}$, where $I_1 \equiv val : \mathsf{int}$, $I_3 \equiv step : \mathsf{void} \to \mathsf{void}$, $I_2 \equiv tmp : I_1$ and $I_4 \equiv run : I_3 \to \mathsf{void}$; and terms $\Delta|x : I_4 \vdash M_1, M_2 : \mathsf{void}$. 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 $N_1 \equiv \mathsf{div}$ and $N_2 \equiv \mathsf{assert}(global.val = \mathsf{halt})$.

```
1  let global = new {_:I1}, aux = new{_:I2} in
2  aux.tmp := new{_:I1};
3  x.run( new{ _:I3;
4           step : λ_. assert (global.val ∈ Q_E);
5                      let mine = new{_:I1}, prev = aux.tmp in
6                      aux.tmp := mine;
7                      mine.val := π1 δ_E(global.val); global.val := π2 δ_E(global.val);
8                      x.run( new{ _:I3;
9                                  step : λ_. assert (global.val ∈ Q_D);
10                                            assert (prev.val = 0 and mine.val ≠ 0);
11                                            global.val := δ_D(global.val, mine.val);
12                                            mine.val := 0;
13                                            if (aux.tmp = mine) then global.val := halt });
14                     N_i });
15 N_i
```

The terms M_i are constructed in such a way that any interaction with them results in a call to $x.run$ (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* $\Delta|x : I_4 \vdash M_1, M_2 : \mathsf{void}$*, 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. $\Delta = \{I_1, I_2, I_3, I_5\}$, where $I_5 \equiv enq : \mathsf{void} \to I_3$. An analogue argument can then be formulated for terms $\Delta|\emptyset \vdash M : I_5$. 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. $\Delta = \{I_1, I_2, I_3, I_4, I_6\}$, where $I_6 \equiv enq : I_4 \to \mathsf{void}$. An analogue argument can then be formulated for terms $\Delta|\emptyset \vdash M : I_6$. 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 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 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.

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

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:

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

Observe that the intersection of L and R is captured by B.

Using the above restrictions, we define IMJ^* to be a fragment of IMJ_f consisting of iterative terms $\Delta|\Gamma \vdash M : \theta$ such that $\mathsf{cod}(\Gamma) \subseteq \mathsf{L}$ and $\theta \in \mathsf{R}$. 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 $\mathsf{let}\, x{=}M\, \mathsf{in}\, M'$), which enforces them to be in B. We write x^{B} to say that x must be in B and not only in L.

$$\begin{aligned} M ::= {} & \mathsf{null} \mid x^{\mathsf{B}} \mid i \mid \mathsf{skip} \mid \mathsf{new}\{\, x : \mathsf{R}\, ; \overrightarrow{\mathsf{m} : \lambda\vec{x}.M}\, \} \mid x = x' \mid a^{\mathsf{B}} \mid M; M' \mid M.\mathsf{f} \\ & \mid \mathsf{if}\, M\, \mathsf{then}\, M'\, \mathsf{else}\, M'' \mid M.\mathsf{f} := M' \mid M = M' \mid M.\mathsf{m}(\vec{M}) \mid (I)M \mid M \oplus M' \\ & \mid \mathsf{while}\, M\, \mathsf{do}\, M' \mid \mathsf{let}\, x = (I)y\, \mathsf{in}\, M \mid y.\mathsf{f} \mid \mathsf{let}\, x = y.\mathsf{m}(\vec{M})\, \mathsf{in}\, M \mid \mathsf{let}\, x = M\, \mathsf{in}\, M' \end{aligned}$$

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-with-store:

$$\mathbb{W} = \{m^{\Sigma} \mid m \in M_{[\![\Delta \mid \Gamma \vdash \theta]\!]} \wedge \Sigma \text{ a store} \wedge \nu(m) \subseteq \mathsf{dom}(\Sigma)\}$$

for given Γ, Δ, θ. 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 $\mathbb{C}_{\mathtt{st}}$ be a finite set of stack symbols. These will vary between different automata. We set:

$$\mathbb{C} = \{\star, \mathtt{nul}\} \cup [0, \textsc{MaxInt}] \qquad\qquad \mathbb{C}_{\mathtt{r}} = \{\, \mathtt{r}_i \mid i \in [1, \mathrm{R}]\, \}$$

$\mathbb{C}$ is the set of constant values that can appear in game moves. $\mathbb{C}_{\mathtt{r}}$ are the constants whose role is to refer to the registers symbolically (e.g. $\mathtt{r}_2$ refers to register number 2).

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

$$\begin{aligned} \mathit{Mov}_{\mathtt{S}} &= \mathit{Val}_{\mathtt{S}}^{*} \cup \{\mathsf{call}\, \mathtt{r}_i.\mathsf{m}.(\vec{x}), \mathsf{ret}\, \mathtt{r}_i.\mathsf{m}(x) \mid x\vec{x} \in \mathit{Val}_{\mathtt{S}}^{+}\} \\ \mathit{Sto}_{\mathtt{S}} &= \mathbb{C}_{\mathtt{r}} \rightharpoonup \mathit{Intfs} \times (\mathit{Flds} \rightharpoonup \mathit{Val}_{\mathtt{S}}) \end{aligned}$$

and $\mathit{Lab}_{\mathtt{S}} = \mathit{Mov}_{\mathtt{S}} \times \mathit{Sto}_{\mathtt{S}}$. For any $x \in \mathit{Val}_{\mathtt{S}} \cup \mathit{Mov}_{\mathtt{S}} \cup \mathit{Sto}_{\mathtt{S}} \cup \mathit{Lab}_{\mathtt{S}}$, we write $\nu_{\mathtt{r}}(x)$ for the set of registers appearing in x. We range over symbolic values by ℓ and variants,

symbolic moves by μ etc, and symbolic stores by S etc. For symbolic labels we may use variants of Φ or, more concretely, μ^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 $Reg = \{\rho : \mathbb{C}_{\mathtt{r}} \stackrel{\cong}{\rightharpoonup} \mathbb{A}\}$ and contains all partial injections from registers to names. Given $\rho \in Reg$ and any $x \in Val_{\mathtt{S}} \cup Mov_{\mathtt{S}} \cup Sto_{\mathtt{S}} \cup Lab_{\mathtt{S}}$, we can define the non-symbolic counterpart of x:

$$\rho(x) = x[\rho(\mathtt{r}_1)/\mathtt{r}_1] \cdots [\rho(\mathtt{r}_{\mathtt{R}})/\mathtt{r}_{\mathtt{R}}]$$

where substitution is defined by induction on the syntax of x. In particular, $\rho(x)$ is undefined if, for some index i, $\mathtt{r}_i \in \nu_{\mathtt{r}}(x)$ but $\rho(\mathtt{r}_i)$ 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 μ^S can be taken and the current register content is ρ, the automaton will perform the transition and accept the letter $\rho(\mu^S) \in \mathbb{W}$.

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:

$$Mov_{\mathtt{S}} = Mov_{\mathsf{push}} \uplus Mov_{\mathsf{pop}} \uplus Mov_{\mathsf{noop}} \qquad Lab_\alpha = \{\mu^S \in Lab \mid \mu \in Mov_\alpha\}$$

where $\alpha \in \{\mathsf{push}, \mathsf{pop}, \mathsf{noop}\}$. We let $Stk = (\mathbb{C}_{\mathtt{st}} \times Reg)^*$ be the set of *stacks*. We shall range over stacks by σ, and over elements of a stack σ by (s, ρ).

Definition 13. The set of ***transition labels*** is $TL = TL_{\mathsf{push}} \uplus TL_{\mathsf{pop}} \uplus TL_{\mathsf{noop}}$ where, for $\alpha \in \{\mathsf{push}, \mathsf{pop}, \mathsf{noop}\}$:

$$TL_\alpha = \{\, (X, \Phi, \phi) \in \mathcal{P}(\mathbb{C}_{\mathtt{r}}) \times Lab_\alpha \times \mathsf{S}_\alpha \mid X \subseteq \nu_{\mathtt{r}}(\Phi) \,\}$$

and $\mathsf{S}_{\mathsf{push}} = \mathbb{C}_{\mathtt{st}} \times \mathcal{P}(\mathbb{C}_{\mathtt{r}})$, $\mathsf{S}_{\mathsf{pop}} = \mathbb{C}_{\mathtt{st}} \times \mathcal{P}(\mathbb{C}_{\mathtt{r}})^2$, $\mathsf{S}_{\mathsf{noop}} = \{\,()\,\}$.

We range over TL by $\nu X.(\Phi, \phi)$, where if $X = \emptyset$ then we may suppress the νX part altogether; if X is some singleton $\{\mathtt{r}_i\}$ then we may shorten $\nu\{\mathtt{r}_i\}$ to $\nu\mathtt{r}_i$. On the other hand, ϕ 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:[3]

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

[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).

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 $\mathcal{A} = \langle Q, q_0, X_0, \delta, F\rangle$ 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;
- $X_0 \subseteq \mathbb{C}_{\mathtt{r}}$ is the set of initially non-empty registers;
- $\delta \subseteq (Q_P \times TL_{PO} \times Q_O) \cup (Q_O \times TL_{OP} \times Q_P) \cup (Q_P \times \mathcal{P}(\mathbb{C}_{\mathtt{r}}) \times Q_P) \cup (Q_O \times (\mathcal{P}(\mathbb{C}_{\mathtt{r}}) \cup (\mathbb{C}_{\mathtt{r}} \overset{\cong}{\to} \mathbb{C}_{\mathtt{r}})) \times Q_O)$ is the transition relation;

where $TL_{PO} = TL_{\mathsf{noop}} \cup TL_{\mathsf{push}}$, $TL_{OP} = TL_{\mathsf{noop}} \cup TL_{\mathsf{pop}}$.

We next explain the semantics of IMJ-automata. Let $\mathcal{A}$ be as above. A ***configuration*** of $\mathcal{A}$ consists of a quadruple $(q, \rho, \sigma, H) \in Q \times Reg \times Stk \times \mathcal{P}_{\mathsf{fn}}(\mathbb{A})$. By saying that $\mathcal{A}$ is in configuration (q, ρ, σ, H) we mean that, currently: the automaton state is q; the register assignment is ρ; the stack is σ; 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, ρ, σ, H). If $q \xrightarrow{\nu X.(\mu^S, \phi)} q'$ then $\mathcal{A}$ will accept an input $m^\Sigma \in \mathbb{W}$ and move to state q' if the following steps are successful.

- If $\mu^S \in Lab_{\mathsf{pop}}$ and $\phi = (s, Y, Z)$ then $\mathcal{A}$ will check whether the stack has the form $\sigma = (s, \rho') :: \sigma'$ with $\mathsf{dom}(\rho') = Y$ and ρ, ρ' being the same in Z and complementary outside it, that is: $\mathsf{dom}(\rho) \cap \mathsf{dom}(\rho') = Z$ and $\rho \cup \rho'$ is a valid assignment. If that is the case, it will pop the top of the stack into the registers, that is, set $\sigma = \sigma'$ and $\rho = \rho \cup \rho'$.
- $\mathcal{A}$ will update the registers in X with fresh names, that is, it will check whether $\mathsf{dom}(\rho) \cap X = \emptyset$ and, if so, it will set $\rho = \rho[\mathtt{r}_{i_1} \mapsto a_1] \cdots [\mathtt{r}_{i_m} \mapsto a_m]$, 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 $a_1, \cdots, a_m \notin \mathsf{cod}(\rho)$ (*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 $H = H \cup \{a_1, \cdots, a_m\}$.
- $\mathcal{A}$ will check whether $m^\Sigma = \rho(\mu^S)$.
- If $\mu^S \in Lab_{\mathsf{push}}$ 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 $\sigma = (s, \rho \restriction Z) :: \sigma$.

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

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

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

Given an initial assignment ρ_0 such that $\mathsf{dom}(\rho_0) = X_0$ and taking $H_0 = \mathsf{cod}(\rho_0)$, the ***language accepted*** by $(\mathcal{A}, \rho_0)$ is

$$\mathcal{L}(\mathcal{A}, \rho_0) = \{w \in \mathbb{W}^* \mid (q_0, \rho_0, \epsilon, H_0) \xrightarrow{w} (q, \rho, \epsilon, H) \wedge q \in F\}.$$

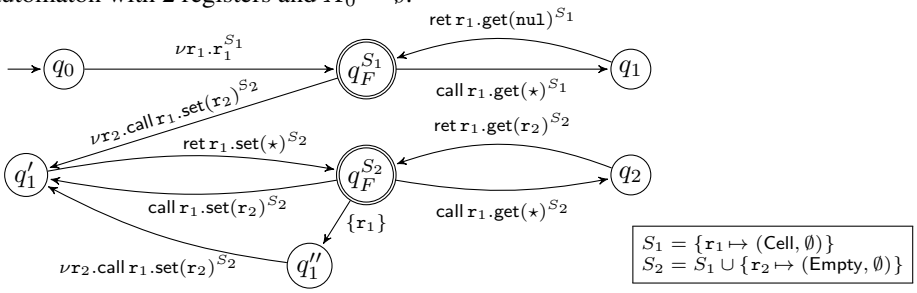
We say that $\mathcal{A}$ is ***deterministic*** if, from any configuration, there is at most one way to accept each input $x \in \mathbb{W}$.

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

Example 15. Consider $\Delta|\emptyset \vdash M_1, M_2 : \mathsf{Cell}$ 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:

$$G_0 = \mathsf{call}\, n.\mathsf{get}(\star)^{\Sigma_0}\ \mathsf{ret}\, n.\mathsf{get}(\mathtt{nul})^{\Sigma_0} \quad G_i = \mathsf{call}\, n.\mathsf{get}(\star)^{\Sigma_i}\ \mathsf{ret}\, n.\mathsf{get}(n_{i-1})^{\Sigma_i} \quad (i > 0)$$
$$L_i = \mathsf{call}\, n.\mathsf{set}(n_i)^{\Sigma_i}\ \mathsf{ret}\, n.\mathsf{set}(\star)^{\Sigma_i}$$
$$\Sigma_i = \{n \mapsto (\mathsf{Cell}, \emptyset)\} \cup \{n_j \mapsto (\mathsf{Empty}, \emptyset) \mid j \in [1, i]\}$$

for any $n, n_1, \cdots, n_i \in \mathbb{A}$ with $n \neq n_j$ ($j \in [1, i]$). We construct the following IMJ-automaton with 2 registers and $X_0 = \emptyset$.



The automaton *represents* plays as above, starting from their second move[4] and matching each Σ_i to a representation $\hat{\Sigma}_i = \Sigma_0 \cup \{n_i \mapsto (\mathsf{Empty}, \emptyset)\}$. The first transition corresponds to the move n^{Σ_0}: the transition label $\nu \mathtt{r}_1.\mathtt{r}_1^{S_1}$ stipulates that the automaton will accept n^{Σ_0}, for some/any fresh name n, and store n in its first register (i.e. register $\mathtt{r}_1$). Observe how the value n, stored in $\mathtt{r}_1$, is invoked in later transitions. For instance, the transition labelled $\mathsf{call}\, \mathtt{r}_1.\mathsf{get}(\star)^{S_1}$ will accept the input $\mathsf{call}\, n.\mathsf{get}(\star)^{\Sigma_0}$. Note also that there are two kinds of transitions involving register $\mathtt{r}_2$. Transitions labelled $\nu \mathtt{r}_2.\mathsf{call}\, \mathtt{r}_1.\mathsf{set}(\mathtt{r}_2)^{S_2}$ set the value of register $\mathtt{r}_2$ to some locally fresh value n'' and accept $\mathsf{call}\, n.\mathsf{set}(n'')^{\Sigma_0 \cup \{n'' \mapsto (\mathsf{Empty}, \emptyset)\}}$ (note how, in the transition from $q_F^{S_2}$ to q_1', the automaton first clears the contents of $\mathtt{r}_2$). On the other hand, the transition $\mathsf{call}\, \mathtt{r}_1.\mathsf{set}(\mathtt{r}_2)^{S_2}$ corresponds to accepting $\mathsf{call}\, n.\mathsf{set}(n')^{\Sigma_0 \cup \{n' \mapsto (\mathsf{Empty}, \emptyset)\}}$ and n' is the current value of register $\mathtt{r}_2$ (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

[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.

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 $\Delta|\Gamma \vdash \theta$ 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 $a \in \mathsf{dom}(\Sigma_{2i}) \setminus \mathsf{dom}(T_{2i})$ then $\Sigma_{2i}(a) = \Sigma_{2i-1}(a)$. The set of all extensions of t is $\mathsf{ext}(t)$.

We can now state our main translation result. Recall that, for each Δ, Γ, we write $P^1_{\Delta|\Gamma}$ for the set of initial moves-with-store in $[\![\Delta|\Gamma \vdash \theta]\!]$. The set of its *initial symbolic moves-with-store* is the finite set: $(\!|\, \Delta|\Gamma \,|\!) = \{\mu_0^{S_0} \in Lab_{\mathsf{S}} \mid \exists \rho_0.\, \rho_0(\mu_0^{S_0}) \in P^1_{\Delta|\Gamma}\}$. We say that a triple $(m^\Sigma, \Phi, \rho) \in P^1_{\Delta|\Gamma} \times (\!|\, \Delta|\Gamma \,|\!) \times \mathit{Reg}$ is *compatible* if $m^\Sigma = \rho(\Phi)$ and $\mathsf{cod}(\rho) = \mathsf{dom}(\Sigma)$, and let $P^{m,\Sigma}_{[\![\Delta|\Gamma\vdash\theta]\!]}$ be the set of plays over $\Delta|\Gamma \vdash \theta$ starting with m^Σ.

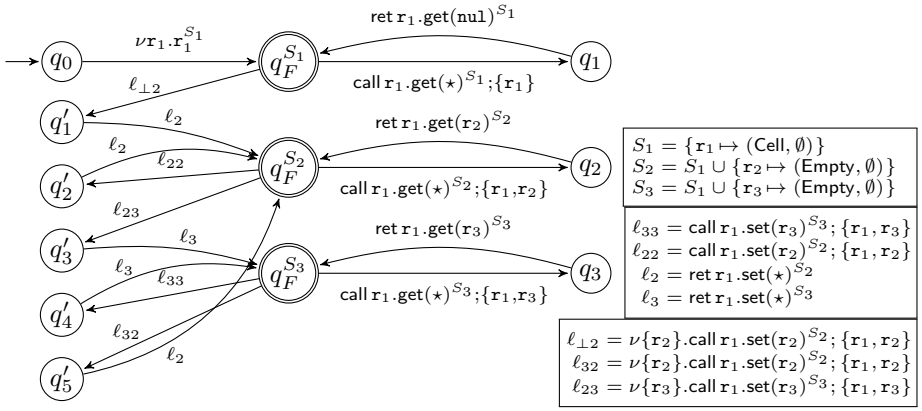
Theorem 17. *Let $\Delta|\Gamma \vdash M : \theta$ be an IMJ^* term. We can effectively define a family of deterministic IMJ-automata $(\!|\, M \,|\!) = \{\, (\!|\, M \,|\!)_\Phi \mid \Phi \in (\!|\, \Delta|\Gamma \,|\!) \,\}$ such that*

$$\bigcup_{w \in \mathcal{L}((\!|\, M \,|\!)_\Phi, \rho)} \mathsf{ext}(m^\Sigma w) = \mathsf{comp}([\![\Delta|\Gamma \vdash M : \theta]\!]) \cap P^{m,\Sigma}_{[\![\Delta|\Gamma\vdash\theta]\!]}$$

for each compatible (m^Σ, Φ, ρ).

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 $\Delta|\emptyset \vdash M_i : \mathsf{Cell}$ $(i = 1, 2)$ from Examples 3,15. 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, q'_2 and q'_5 can be evidently unified (q'_3, q'_4 too). There is also a symmetry between $q_F^{S_2}$ and $q_F^{S_3}$.

Let now $\Delta|\Gamma \vdash M_1, M_2 : \theta$ be IMJ^* terms. The previous theorem provides us with deterministic IMJ-automata $\mathcal{A}_i = (\!|\, M_i \,|\!)_\Phi$ $(i = 1, 2)$ representing the complete plays of $[\![M_i]\!]$ which start with m^Σ, for each compatible triple (m^Σ, Φ, ρ). 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^Σ, Φ, ρ).

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 $m'^{\Sigma'}$:

- if $\mathcal{A}_1$ and $\mathcal{A}_2$ both accept $m'^{\Sigma'}$ (modulo extensions) then $\mathcal{B}$ accepts $m'^{\Sigma'}$ and proceeds its joint simulation of $\mathcal{A}_1, \mathcal{A}_2$;
- if, say, $\mathcal{A}_1$ accepts $m'^{\Sigma'}$ 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 $m'^{\Sigma'}$). 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 $\Delta|\Gamma \vdash M_1, M_2 : \theta$ and (m^Σ, Φ, ρ) as above, we can effectively construct an IMJ-automaton $\mathcal{B}$ such that $\mathcal{L}(\mathcal{B}, \rho) = \emptyset$ iff $\mathsf{comp}([\![\Delta|\Gamma \vdash M_1 : \theta]\!]) = \mathsf{comp}([\![\Delta|\Gamma \vdash M_2 : \theta]\!])$.*

As variants of *pushdown fresh-register automata*, IMJ-automata have decidable emptiness problem [10]. Moreover, the number of compatible triples (m^Σ, Φ, ρ) is bounded with respect to Γ, Δ 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 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 Δ is ***first-order*** just if $I.m$ has type of shape $(\mathsf{G}, \ldots, \mathsf{G}) \to \mathsf{G}$. Otherwise we shall say that it is ***higher-order***. Fix a term $\Delta|\emptyset \vdash P : \mathsf{void}$. We say that P is 1***-recursive*** just if, whenever there is a cycle $(I_1, m_1), \ldots, (I_k, m_k)$ 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 Δ have only* G*-valued fields and $\Delta|\emptyset \vdash M : \mathsf{void}$ 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.