Game Semantic Analysis of Equivalence in IMJ

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

[1] University of Warwick [2] Queen Mary University of London

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. This is made precise in Appendix A.

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, but is also formalised in Appendix A. 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.

We write $\mathit{EStacks}$ for the set of *evaluation stacks*, which are lists of evaluation contexts. The transition relation is defined on terms within a state and evaluation stack, that is, on triples $(S, M, F) \in \mathit{States} \times \mathit{Terms} \times \mathit{EStacks}$ and is presented in Appendix A.

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 $(\emptyset, M, \epsilon) \rightarrow^* (S, \mathsf{skip}, \epsilon)$.

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 $\{\ \_ \colon \mathsf{Cell};$
$get\ :\ \lambda\_.\ v.val\,,$
$set\ :\ \lambda y.$ 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 $\{\ \_ \colon \mathsf{Cell};$
$get\ :\ \lambda\_.$ 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 $v.val{:=}y; w.val{:=}y\ \}$

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*

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

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 Case 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 observational 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.

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

[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.

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

- $\Delta(I).\mathsf{f} = \theta'$ 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

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

and $M_{[\![\theta]\!]} = Val_\theta$. A ***move-with-store*** for $\Delta|\Gamma \vdash \theta$ is pair of a move and a store, and is 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 are 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.

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 : $\lambda\_.$ 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*: $\lambda\_.$ 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 Observation 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, given a queue machine $\mathcal{Q}$, 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. *Observational 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 refer the reader to the Appendix for details, and 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 of Lemma 30 can then be shown 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 of Lemma 30 can then be shown 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 the grammar given below. Note that 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.

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

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 $\mathbb{W}$ which contains game 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.

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:

[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).

– 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 (a more concise formal definition is given in Appendix E). 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

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

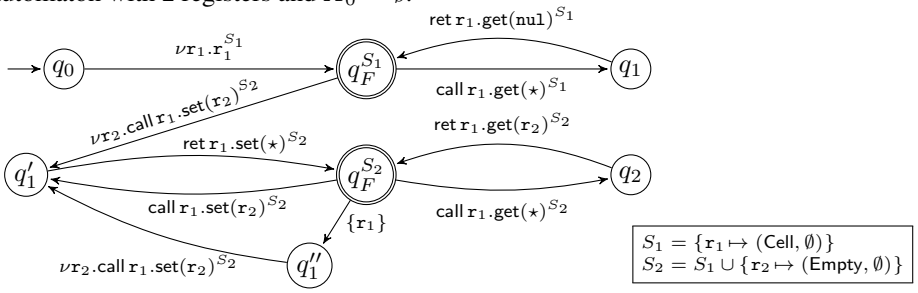
We say that $\mathcal{A}$ is ***deterministic*** if, from any configuration, there is at most one way to accept each input $x \in \mathbb{W}$.

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:

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

for any $n, n_1, \cdots, n_i \in \mathbb{A}$ with $n \neq n_j$ ($j \in [1, i]$). We construct the following 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 we observed 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

[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.

by definition 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.

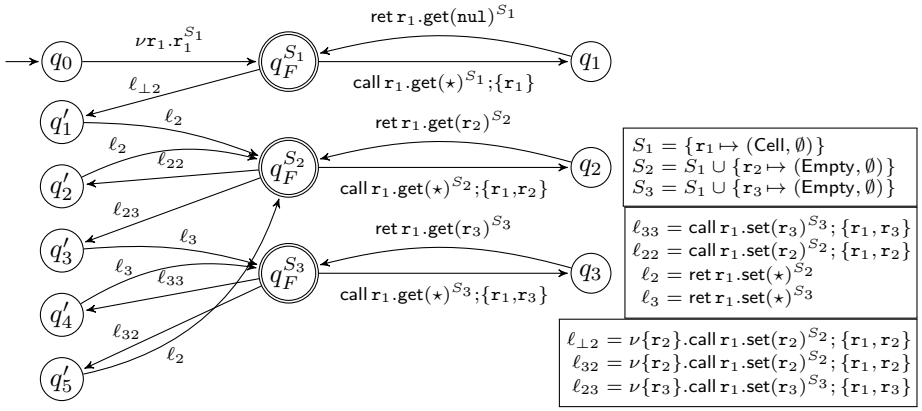
Theorem 17. *Let $\Delta|\Gamma \vdash M : \theta$ be an IMJ^* term. We can effectively define a family of deterministic IMJ-automata $(\!|\, M \,|\!) = \{\, (\!|\, M \,|\!)_\Phi \mid \Phi \in (\!|\, \Delta|\Gamma \,|\!) \,\}$ such that*

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

for each compatible (m^Σ, Φ, ρ).

The construction is presented in detail in Appendices D,E and 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.

A Syntax and operational semantics of IMJ$_f$

To formalise the *recursive types restriction* we require that interface tables Δ be well formed, which is to say that $\Delta \vdash I$ is provable, for each I defined in Δ, by the following inference rules

$$\frac{}{\Delta \vdash \mathsf{void/int}} \qquad \frac{\Delta \vdash \theta\ (\forall \theta \in \Theta)}{\Delta \vdash I_1} \qquad \frac{\Delta \vdash \theta\ (\forall \theta \in \Theta) \quad \Delta \vdash I}{\Delta \vdash I_2}$$

given $I_1 \equiv \Theta \in \Delta$ and $I_2\langle I \rangle \equiv \Theta \in \Delta$. In this system, we write $\theta \in \Theta$ as a short-hand for the existence of some field $(\mathsf{f} : \theta') \in \Theta$ or some method $(\mathsf{m} : \vec{\theta} \to \theta') \in \Theta$ such that θ is θ' or otherwise belongs to the vector $\vec{\theta}$.

The subtyping relation is formalised as follows. Given a table Δ, we define $\Delta \vdash \theta_1 \leq \theta_2$ by the rules:

$$\frac{}{(I\langle I' \rangle \equiv \Theta), \Delta \vdash I \leq I'} \qquad \frac{}{\Delta \vdash \theta \leq \theta} \qquad \frac{\Delta \vdash \theta_1 \leq \theta_2 \leq \theta_3}{\Delta \vdash \theta_1 \leq \theta_3}$$

We might omit Δ from subtyping judgements for economy.

Given an interface table Δ such that $I \in \mathsf{dom}(\Delta)$, we let the default heap configuration of type I be $V_I = \{\mathsf{f} : v_\theta \mid \Delta(I).\mathsf{f} = \theta\}$, where $v_{\mathsf{void}} = \mathsf{skip}$, $v_{\mathsf{int}} = 0$ and $v_I = \mathsf{null}$. The operational semantics is defined by the following system of rules:

$$\begin{aligned}
(S, E[i \oplus i'], F) &\longrightarrow (S, E[\ulcorner i \oplus i' \urcorner], F) \\
(S, E[\mathsf{skip}; M], F) &\longrightarrow (S, E[M], F) \\
(S, E[\mathsf{if}\ 0\ \mathsf{then}\ M\ \mathsf{else}\ M'], F) &\longrightarrow (S, E[M], F) \\
(S, E[\mathsf{if}\ i\ \mathsf{then}\ M\ \mathsf{else}\ M'], F) &\longrightarrow (S, E[M'], F)\ \text{ when } i > 0 \\
(S, E[(I)a], F) &\longrightarrow (S, E[a], F)\ \text{ when } S(a) : I' \wedge I' \leq I \\
(S, E[(I)\mathsf{null}], F) &\longrightarrow (S, E[\mathsf{null}], F) \\
(S, E[a = a'], F) &\longrightarrow (S, E[0], F)\ \text{ when } a \neq a' \\
(S, E[a = a], F) &\longrightarrow (S, E[1], F) \\
(S, E[a.\mathsf{f} := v], F) &\longrightarrow (S[a \mapsto S(a)[\mathsf{f} \mapsto v]], E[\mathsf{skip}], F) \\
(S, E[a.\mathsf{f}], F) &\longrightarrow (S, E[S(a).\mathsf{f}], F) \\
(S, E[\mathsf{let}\ x = v\ \mathsf{in}\ M], F) &\longrightarrow (S, E[M[v/x]], F) \\
(S, E[a.\mathsf{m}(\vec{v})], F) &\longrightarrow (S, M_{a.\mathsf{m}}[\vec{v}/\vec{x}], E :: F) \\
(S, v, E :: F) &\longrightarrow (S, E[v], F) \\
(S, E[\mathsf{new}\{z{:}\,I; \mathcal{M}\}], F) &\longrightarrow (S \uplus \{(a, I, (V_I, \mathcal{M}[a/z]))\}, E[a], F) \\
(S, E[\mathsf{while}\ 0\ \mathsf{do}\ M], F) &\longrightarrow (S, E[\mathsf{skip}], F) \\
(S, E[\mathsf{while}\ i\ \mathsf{do}\ M], F) &\longrightarrow (S, M, E[_\,; \mathsf{while}\ i\ \mathsf{do}\ M] :: F)
\end{aligned}$$

B Proofs from Section 3

B.1 Proof of Theorem 4

Definition 23. Let $\mathcal{A}$ be a finite alphabet. A *queue machine* $\mathcal{Q}$ over $\mathcal{A}$ is a tuple of shape $\langle Q, Q_E, Q_D, q_I, \delta_E, \delta_D \rangle$ where $Q = Q_E \uplus Q_D$ is a finite set of states with $q_I \in Q_E$ initial, $\delta_E : Q_E \to Q \times \mathcal{A}$ is the enqueuing function and $\delta_D : Q_D \times \mathcal{A} \to Q$ is the dequeueing function.

For simplicity, in what follows we shall assume that $\mathcal{A} = \{1, \ldots, k\}$ and $Q = \{0, \ldots, m\}$ with $q_I = 0$. We also assume that the queue is initially empty and the machine is deemed to halt once the queue becomes empty again. It is well known that the halting problem for such machines is undecidable.

Proof. In both cases we reduce from the halting problem for queue machines. Given a queue machine $\mathcal{Q}$ as specified above, we construct terms M such that $M\Downarrow$ if and only if $\mathcal{Q}$ halts. The terms M are shown below:

```
M ≡
new { x: P;
 main: λ_.
   x.queue := new { _: N;
         next: λ_. skip };
   while (x.halt = 0) do
     M_{x.state}
}.main()

ENQ(i) ≡
new { z: T;
 run: λ_.
    z.n := x.queue;
    x.queue := new { _: N;
      next: λ_. x.queue := z.n };
    x.queue.v := π2(δE(i))
}.run();
x.state := π1(δE(i))

DEQ(i) ≡
x.tmp := x.queue;
x.prev := x.queue;
while (x.queue.v) do
   { x.prev := x.queue;
     x.queue.next() };
x.state := δD(i, x.prev.v);
x.prev.v := 0;
if (x.tmp.v = 0) then
                 x.halt := 1;
x.queue := x.tmp
```

```
M ≡
new { x: P;
 main: λh p w.
    let cur = new {_: V}
    in {
    cur.v := h;
    while (x.halt = 0) do
      M_{x.state} }
}.main(0, null, null)

ENQ(i) ≡
x.state := π1(δE(i));
x.main( π2(δE(i)), cur,
  new { _: N;
    next: λ_.
      if p.v then w.next()
             else cur
  })


DEQ(i) ≡
let z =
   if p.v then w.next()
          else cur
in {
x.state := δD(i, z.v );
z.v = 0;
if (z = cur) then
          x.halt := 1
}
```

We use $M_{x.state}$ as shorthand for the conditional expression that branches on *x.state* and selects branch $ENQ(x.state)$ if *x.state* $\in Q_E$ and $DEQ(x.state)$ if *x.state* $\in Q_D$. In each case, the encoding builds an unbounded chain of objects which are linked through the definition of method implementations. For example, in Case 1 of Theorem 4, the simulated queue is built from a chain of objects of type N. The link is formed by capturing an existing object in a closure which forms part of the definition of the method $next : \mathsf{void} \to \mathsf{void}$. The terms have type void and, in the first case, rely on the following interfaces:

$$\begin{aligned} N \equiv\ & v : \mathsf{int}, next : \mathsf{void} \to \mathsf{void} \\ P \equiv\ & queue : N, prev : N, tmp : N, \\ & state : \mathsf{int}, halt : \mathsf{int}, main : \mathsf{void} \to \mathsf{void} \\ T \equiv\ & n : N, run : \mathsf{void} \to \mathsf{void} \end{aligned}$$

and, in the second case, they rely on the interfaces:

$$\begin{aligned} V \equiv\ & v : \mathsf{int} \qquad\qquad N \equiv\ next : \mathsf{void} \to V \\ P \equiv\ & state : \mathsf{int}, halt : \mathsf{int}, main : (\mathsf{int},\ V,\ N) \to \mathsf{void} \end{aligned}$$

□

By passing parameters/results through fields, it is possible to simplify the types of $main$, $next$ to $N \to \mathsf{void}$, $\mathsf{void} \to \mathsf{void}$ respectively. This alternative encoding of a queue machine given in Figure 2 uses types:

$$\begin{aligned} P \equiv\ & arg_1 : \mathsf{int}, arg_2 : C, ret : C \\ & state : \mathsf{int}, halt : \mathsf{int}, main : N \to \mathsf{void} \\ C \equiv\ & v : \mathsf{int} \\ N \equiv\ & next : \mathsf{void} \to \mathsf{void} \end{aligned}$$

B.2 Proof of Theorem 6

We start with a formal definition of visible state.

Definition 24. Let (S, M, F) be a configuration. We define the *visible names* of M, written $S@M$, to be the set of names that are S-reachable from M. Explicitly, we set $S@M = S^*(\nu(M))$ and, for any set of names A, $S^*(A) = \bigcup_{i\in\omega} S^i(A)$ with:

$$S^0(A) = A, \qquad S^{i+1}(A) = S^i(A) \cup \{\, \nu(S(a)) \mid a \in S^i(A) \,\}.$$

Given $\Delta|\emptyset \vdash M : \mathsf{void}$, we say that M has ***depth-bounded visible state*** if there is some $k \in \omega$ such that, for all triples (S, M', F), if $(\emptyset, M, \epsilon) \longrightarrow^* (S, M', F)$ then $|S@M'| \leq k$.

Our aim in this section is to show that iterative terms $\Delta|\emptyset \vdash M : \mathsf{void}$ (with fields in Δ restricted to G) have depth-bounded visible state. Our argument proceeds in three parts. First we define an abstract operational semantics for terms. For simplicity, we assume that terms do not include any occurrence of let since, as remarked in the definition of IMJ_f, this can be encoded using object creation and calls. The abstract semantics

```
ENQ(i) ≡
  x. state  := π1(δE(i));
  x.arg1 := π2(δE(i));
  x.arg2 := cur;
  main(new {_:N; next : λ_. if p.v = 0 then x. ret := cur else w.next()})
```

```
DEQ(i) ≡
  if cur.v = 0 then
    x. halt := 1
  else
    if p.v = 0 then x. ret := cur else w.next()
    let z = x. ret .v in
    if z = 1 then
      x. state := δD(i, 1)
    else if z = 2 then
      ...
    else if z = k then
      x. state := δD(i, k)
    else skip
    x. ret .v := 0
```

```
M ≡
  new {
    x: P;
    main: λw.
      let p = x.arg2 in
      let cur = new {_: C;} in
      cur.v := x.arg1;
      while x. halt = 0 do
        if x. state = 0 then
          M0
        else if x. state = 1 then
          ...
        else if x. state = m then
          Mm
        else skip
  }.main(null)
```

Fig. 2. Encoding of queue machine using recursion with simpler types.

ignores the particular value of objects that do not carry methods and integers since they are both inheritantly bounded in finitary IMJ. Next, we show that the abstract semantics is a sound approximation of the usual operational semantics and that the abstract operational semantics of a term which does not contain while constructs is strongly normalising. Hence, there is always some bound on the size of the visible state during transitions. Finally, we show that, for the purpose of reaching new visible states, occurrences of while in a term can be eliminated.

We define an abstract operational semantics for terms.

Definition 25. The ***abstract values*** are defined as: $AVals \ni v_\star ::= \mathsf{skip} \mid \star \mid a$. The default integer value and the default value for interface types that do not carry methods is $\star$. Interfaces that do carry methods default to null, as before. We write $V_I^\star$ for the heap configuration of type I that has all fields initialised to default abstract values. Let us say a value is ground if it is of type integer or an interface type that does not carry methods. Given a closed term M, we write $\widehat{M}$ for the term we obtain from M by replacing all ground values in M by default abstract ones. We define ***abstract terms*** to be terms that may only feature abstract values, and similarly for states and evaluation contexts and stacks. The rules for the abstract operational semantics are given in Figure B.2. Finally, we write $S@_m M$ for the restriction of $S@M$ to method-carrying object names.

$$
\begin{aligned}
(S, E[\star \oplus \star], F) &\longrightarrow (S, E[\star], F) \\
(S, E[\mathsf{skip}; M], F) &\longrightarrow (S, E[M], F) \\
(S, E[\mathsf{if}\ \star\ \mathsf{then}\ M\ \mathsf{else}\ M'], F) &\longrightarrow (S, E[M'], F) \\
(S, E[\mathsf{if}\ \star\ \mathsf{then}\ M\ \mathsf{else}\ M'], F) &\longrightarrow (S, E[M], F) \\
(S, E[(I)v_\star], F) &\longrightarrow (S, E[v_\star], F) \\
(S, E[v_\star = v'_\star], F) &\longrightarrow (S, E[\star], F) \\
(S, E[v_\star.\mathsf{f}], F) &\longrightarrow (S, E[\star], F) \\
(S, E[v_\star.\mathsf{f} := v'_\star], F) &\longrightarrow (S, E[\mathsf{skip}], F) \\
(S, E[a.\mathsf{m}(\vec{v_\star})], F) &\longrightarrow (S, M_{a.\mathsf{m}}[\vec{v_\star}/\vec{x}], E :: F) \\
(S, E[\mathsf{while}\ \star\ \mathsf{do}\ M], F) &\longrightarrow (S, E[\mathsf{skip}], F) \\
(S, E[\mathsf{new}\{\, z : I_{\neg m}; \mathcal{M}\,\}], F) &\longrightarrow (S, E[\star], F) \\
(S, v_\star, E :: F) &\longrightarrow (S@F[E[v_\star]], E[v_\star], F) \\
(S, E[\mathsf{new}\{\, z : I; \mathcal{M}\,\}], F) &\longrightarrow (S \uplus \{(a, I, (V_I^\star, \mathcal{M}[a/z]))\}, E[a], F)\ \ \{\text{if } I \text{ has methods}\} \\
(S, E[\mathsf{while}\ \star\ \mathsf{do}\ M], F) &\longrightarrow (S, M, (E[\_; \mathsf{while}\ \star\ \mathsf{do}\ M]) :: F)
\end{aligned}
$$

Fig. 3. Abstract operational semantics of IMJ_f ($S(a).\mathsf{m} = \lambda\vec{x}.M_{a.\mathsf{m}}$ and $I_{\neg m}$ has no methods).

The abstract semantics simulates the concrete semantics with respect to the method-carrying objects present in visible states.

Lemma 26. *Given iterative $\Delta|\emptyset \vdash M : \mathsf{void}$ without method-carrying fields, if $(\emptyset, M, \epsilon) \longrightarrow^* (S, M', F)$ then there is some abstract state S' such that $S@_m F[M'] = S'@\widehat{F}[\widehat{M'}]$ and $(\emptyset, \widehat{M}, \epsilon) \longrightarrow^* (S', \widehat{M'}, \widehat{F})$.*

Proof. The proof is by induction on the length of the transition sequence. The base case is trivial, so assume that the conclusion holds of $(S,\ M,\ F)$, i.e. there is an abstract state S'' such that $S@_m F[M] = S''@\widehat{F}[\widehat{M}]$. Let $(S,\ M,\ F) \longrightarrow (S',\ M',\ F')$. We distinguish cases based on the rule justifying the transition. The only interesting case is that for the creation of objects carrying methods, which we give in detail.

When the transition is by evaluating an object creation expression, so that $M = E[\mathsf{new}\,\{z : I;\ \mathcal{M}\}]$ where $\mathcal{M}$ is non-empty, then $\widehat{M}$ is $\widehat{E}[\mathsf{new}\,\{z : I;\ \widehat{\mathcal{M}}\}]$. The successor to the concrete configuration has state S with $(a,\ I,\ (V_I^\star,\ \mathcal{M}[a/z]))$ added. The successor to the abstract configuration is S'' with $(a,\ I,\ (V_I^\star,\ \widehat{\mathcal{M}}[a/z]))$ added, let us call it S'''. Now, assume that $b \in S'@_m F[M']$ is a method carrying object. If $a = b$ then also $a \in S'''@\widehat{F}[\widehat{M'}]$ since $a \in \nu(\widehat{M'})$. If $a \neq b$ and $b \in S'@_m F$ then the result follows by assumption. Otherwise, $a \neq b$ and b is reachable through $\mathcal{M}[a/z]$; since all method carrying objects in $\mathcal{M}[a/z]$ are also present in $\widehat{\mathcal{M}}[a/z]$, it follows from the

assumption that b is reachable through $\widehat{\mathcal{M}}[a/z]$. The other inclusion is similar (observe that no transition of the abstract semantics introduces more names than the corresponding transition of the concrete semantics). □

When the term to be evaluated does not contain any while constructors, since there is also no recursion there is no capacity for non-termination whatsoever. To carry out the proof we will find it useful to define an *instance* of an abstract term M, which is any term M' obtained from M by replacing every free variable by an abstract value of the same type.

Lemma 27. *Given iterative $\Delta|\emptyset \vdash M : \mathsf{void}$ without method-carrying fields, if M contains no* while *constructors then the reduction tree of $(\emptyset, \widehat{M}, \epsilon)$ has finite depth.*

Proof. Since M is iterative, its finite method dependency graph is acyclic. Given any method $I.m$ we set its measure $\#I.m$ to be the largest distance from $(I,\ m)$ to a leaf in this graph. We prove that, for any method $I.m$ and any body N of an implementation of $I.m$ in M, and any state S, the reduction graph of $(S,\ N,\ \epsilon)$ is finite. The proof is by induction on the measure.

If $\#I.m$ is 0 then N contains no calls. Since, in the absence of while constructs, calls and returns, every transition of the abstract semantics strictly decreases the size of the term, it follows that $(S,\ N,\ \epsilon)$ is strongly normalising.

If $\#I.m > 0$ then N may contain a call, so that a configuration $(S',\ E[a.m'(\vec{\star})],\ \epsilon)$ is reachable with a of some interface type J and $S'(a).m'$ of shape $\lambda\vec{x}.\, N'$. In that case, it follows from the acyclicity of the dependency graph that $J.m'$ is strictly closer to a leaf than $I.m$ and hence the induction hypothesis guarantees that every reduction of $(S',\ N[\vec{\star}/\vec{x}],\ \epsilon)$ reaches some value of shape $(S'',\ v_\star,\ \epsilon)$. Consequently, also $(S',\ N[\vec{\star}/\vec{x}],\ E)$ will reach $(S'',\ v_\star,\ E)$ and necessarily transition to $(S'',\ E[v_\star],\ \epsilon)$ with $E[v_\star]$ a strictly smaller term than $E[a.m'(\vec{\star})]$.

To see that the statement of the lemma is true, observe that $\widehat{M}$ may be reformulated as a method body by constructing the term $\mathsf{new}\,\{z : P;\ run : \lambda x.\, \widehat{M}\}.run()$ for some new interface $P \equiv (run : \mathsf{void} \to \mathsf{void})$ and variables z and x. □

All that remains is to observe that, for the purposes of reaching larger and larger visible states, while constructs can be eliminated.

Corollary 28. *Every iterative $\Delta|\emptyset \vdash M : \mathsf{void}$ using* G*-valued object fields has depth bounded visible state.*

Proof. We construct $\widetilde{M}$ by taking $\widehat{M}$ and replacing every occurrence of $\mathsf{while}\, P\, \mathsf{do}\, M'$ with $\mathsf{if}\, P\, \mathsf{then}\, M'\, \mathsf{else}\, \mathsf{skip}$. Then, by Lemma 27, the reduction tree of $(\emptyset, \widetilde{M}, \epsilon)$ is finite-depth. Hence, there is a bound k such that, for all $(\emptyset, \widetilde{M}, \epsilon) \longrightarrow^* (S, M', F)$, $|S@M'| \leq k$. Now observe that, in the abstract operational semantics, whenever we have a sequence $(S, E[\mathsf{while}\ \star\ \mathsf{do}\, M'], F) \longrightarrow^* (S', E[\mathsf{while}\ \star\ \mathsf{do}\, M'], F)$ which does not contain a reduction of $\mathsf{while}\ \star\ \mathsf{do}\, M'$ to skip, it is the case that $S' \subseteq S$. Hence, in the reduction tree of $(\emptyset, \widehat{M}, \epsilon)$, we can short-circuit while-loops and show that $\widehat{M}$ too is visibly bounded with the same bound k. Then, by Lemma 26, we have that M is visibly bounded. Note, however, that the particular bound for M can be larger than the

bound for $\widehat{M}$ as, in the concrete semantics, we include interface fields and their chains of values (which have bounded length due to the non-recursive types restriction). □

Corollary 29. *If $\Delta|\emptyset \vdash M : \mathsf{void}$ is an iterative term and fields in Δ belong to G then $M\Downarrow$ is decidable.*

Proof. During any evaluation of M, methods attached to objects in the visible state are always instances of subterms of M. Since there are only finitely many subterms and since M has depth-bounded visible state, it follows that there is a bound on the number of shapes of visible state, modulo the specific choice of object names. Hence, we follow [3] in representing the computation as a pushdown system. □

C Undecidability of Observational Equivalence

Case 1

Next we explain how to represent runs of queue machines through plays over $I_4 \vdash \mathsf{void}$. We want each such play to start with $o^{\{o\mapsto I_4\}}\,\mathsf{call}\,o.run(n)^{\Sigma_0}$ where $\Sigma_0 = \{o \mapsto I_4, n \mapsto I_3\}$. This initial segment plays an auxiliary role and does not correspond to any queue operation. It will be followed by other segments (corresponding to enqueuings and dequeuings) that occur in the play in the same order as they occur in a machine run. Specifically, to represent a single queue operation, we use two-move segments consisting of an O-move and a P-move as detailed below.

- The ith enqueuing is represented by the segment $\mathsf{call}\,n.step(\star)^{\Sigma_{i-1}}\,\mathsf{call}\,o.run(n_i)^{\Sigma_i}$, where $\Sigma_i = \Sigma_{i-1} \uplus \{n_i \mapsto I_3\}$. Note that we require that each n_i be fresh (in order to be able to identify enqueuings with n_i).
- The ith dequeuing is represented by $\mathsf{call}\,n_i.step(\star)^{\Sigma_j}\;\;\mathsf{ret}\,n_i.step(\star)^{\Sigma_j}$, where j is the number of enqueuings preceding the ith dequeuing in the machine run.

Note that the moves corresponding to enqueuings are both calls. Consequently, plays interpreting runs according to the above recipe will not be complete. However, they can be completed by adding j segments of the form $\mathsf{ret}\,o.run(\star)^{\Sigma_j}\,\mathsf{ret}\,n.step(\star)^{\Sigma_j}$, where j is the number of enqueuings, and, finally, the segment $\mathsf{ret}\,o.run(\star)^{\Sigma_j}\;\star^{\Sigma_j}$.

Given a queue machine $\mathcal{Q}$, let $R_\mathcal{Q}$ be the set of plays containing representations of transition sequences of $\mathcal{Q}$. If $\mathcal{Q}$ halts, define $R^+_\mathcal{Q}$ to be $R_\mathcal{Q}$ extended with the completion of the play corresponding to the accepting run along with its prefixes. If $\mathcal{Q}$ does not halt, let $R^+_\mathcal{Q}$ be simply $R_\mathcal{Q}$.

Lemma 30. *For any queue machine $\mathcal{Q}$, there exist $\Delta|x : I_5 \vdash M_1, M_2 : \mathsf{void}$ such that $\mathsf{comp}([\![M_1]\!]) = R_\mathcal{Q}$ and $\mathsf{comp}([\![M_2]\!]) = R^+_\mathcal{Q}$.*

Proof. Let $\mathcal{Q} = \{Q, Q_E, Q_D, \delta, q_I, \delta_E, \delta_D\}$. W.l.o.g. let us assume that int is large enough to accommodate $(Q_E \uplus Q_D \uplus \{\mathsf{halt}\}) \cup (\mathcal{A} \uplus \{0\})$. If this is not true, we use fields to achieve the required storage capacity. For $i \in \{1, 2\}$, let M_i be defined by the term in Figure 4, where $N_1 \equiv \mathsf{div}$ and $N_2 \equiv \mathsf{assert}(\mathit{global.val} = \mathsf{halt})$. The terms use *global* to keep track of the state of $\mathcal{Q}$. The required behaviour in the starting segment is achieved by the outermost occurrence of *x.run(...)*.

The outermost definition of *step* is responsible for handling interactions that correspond to enqueuing. Asserting $(\mathit{global.val} \in Q_E)$ ensures that $\mathsf{call}\,n.step(\star)^{\Sigma_{i-1}}$ will

```
let global = new {_:I1} in
let aux = new{_:I2} in
aux.tmp := new{_:I1};
x.run( new{ _:I3;
    step: λ_.
      assert (global.val ∈ QE);
      let mine = new{_:I1} in
      let prev = aux.tmp in
       aux.tmp := mine;
       mine.val := π1 δE(global.val);
       global.val := π2 δE(global.val);
       x.run( new{ _:I3;
          step: λ_.
            assert (global.val ∈ QD);
            assert (prev.val = 0 and mine.val ≠ 0);
            global.val := δD(global.val, mine.val);
            mine.val := 0;
            if (aux.tmp = mine) then global.val := halt });
      Ni });
Ni
```

Fig. 4. Encoding of the queue.

trigger a response only if $\mathcal{Q}$ is ready to enqueue. The expected response $\mathsf{call}\, o.run(n_i)^{\Sigma_i}$ corresponds to the innermost occurrence of *x.run(...)*. Note that the argument to *run* is a newly created object. The object has access to two local variables, *mine* and *prev*, which are essentially its private fields. *mine* is used to store the item that is being enqueued, while *prev* is used to store the reference to the copy of *mine* used in the previous enqueuing. This is done by passing *mine* from one enqueuing to the next using the global variable *aux*. The $prev$ variable is crucial to maintaining the queue discipline, as it allows us to gain information about the previous item that was enqueued. In particular, we can find out whether the item has already been dequeued.

Dequeuing is handled by the inner definition of *step*. The first assertion, namely ($global.val \in Q_D$), guarantees that when the environment plays $\mathsf{call}\, n_i.step(\star)^{\Sigma_j}$, the program can respond only if $\mathcal{Q}$ is ready to dequeue. The second assertion terminates the interaction if n_i does not correspond to the first item that has not been dequeued: *prev.val = 0* means that the preceding item has been dequeued and *mine.val* $\neq$ *0* means that the item corresponding to n_i has not been dequeued. If the assertions succeed, *global* is updated according to δ_D and the local content of *mine* is set to 0 to mark the item as dequeued. If the dequeued item was the last one added (*aux.tmp = mine*), the *global* variable is set to a special value halt. Note that, since all operations inside the innermost *step* are local, $\mathsf{call}\, n_i.step(\star)^{\Sigma_j}$ will trigger $\mathsf{ret}\, n_i.step(\star)^{\Sigma_j}$ as long as the assertions do not fail.

The N_i terms make sure that $\mathsf{ret}\, o.run(\star)^{\Sigma_j}$ will terminate the play unless $i = 2$ and $\mathcal{Q}$ has terminated. This accounts for the difference between $R_{\mathcal{Q}}$ and $R^+_{\mathcal{Q}}$ in the lemma. □

Note that if $\mathcal{Q}$ does not halt, M_1, M_2 will generate the same plays, i.e. $R_{\mathcal{Q}}$, none of which will be complete. However, if $\mathcal{Q}$ halts, $[\![M_2]\!]$ will contain more plays, namely those in $R^+_{\mathcal{Q}} \setminus R_{\mathcal{Q}}$. In particular, one of them will be complete. Thus, $M_1 \cong M_2$ if and only $\mathcal{Q}$ does not halt. Consequently, $\cong$ is undecidable in this case.

Case 2

The encoding relies on the following interfaces.

$$\begin{aligned} I_1 &\equiv val : \mathsf{int} \\ I_2 &\equiv tmp : I_1 \\ I_3 &\equiv step : \mathsf{void} \to \mathsf{void} \\ I_5 &\equiv enq : \mathsf{void} \to I_3 \end{aligned}$$

Next we explain how we represent runs of queue machines through plays over $\Delta, \emptyset \vdash I_5$. Each such play will start with the segment $\star^{\emptyset}\, n^{\Sigma_0}$, where $\Sigma_0 = \{n : I_5\}$. The two kinds of queue operations will be represented by two-move segments consisting of an O-move and a P-move, as detailed below.

- The ith enqueuing is represented by the segment

$$\mathsf{call}\, n.enq(\star)^{\Sigma_{i-1}} \quad \mathsf{ret}\, n.enq(n_i)^{\Sigma_i},$$

  where $\Sigma_i = \Sigma_{i-1} \uplus \{n_i \mapsto I_3\}$, i.e. we require that each enqueuing corresponds to a fresh name n_i.
- The ith dequeueing will correspond to the segment

$$\mathsf{call}\, n_i.step(\star)^{\Sigma_j}\, \mathsf{ret}\, n_i.step(\star)^{\Sigma_j},$$

  where j is the number of enqueuings preceding the ith dequeuing.

Given a queue machine $\mathcal{Q}$, let $R_{\mathcal{Q}}$ be the set containing the sequences of segments corresponding to sequences of transitions made by $\mathcal{Q}$. Define $R^-_{\mathcal{Q}}$ to be $R_{\mathcal{Q}}$ except that, if $\mathcal{Q}$ halts, take away the play corresponding to the last dequeuing operation. Note that the plays corresponding to sequences of queue operations are all complete in this case.

Lemma 31. *For any queue machine $\mathcal{Q}$, there exist $\Delta|\emptyset \vdash M_1, M_2 : I_5$ such that* $\mathsf{comp}([\![M_1]\!]) = R_{\mathcal{Q}}$ *and* $\mathsf{comp}([\![M_2]\!]) = R^-_{\mathcal{Q}}$.

Proof. Let $\mathcal{Q} = \{Q, Q_E, Q_D, \delta, q_I, \delta_E, \delta_D\}$. W.l.o.g. let us assume that int is large enough to accommodate $(Q_E \uplus Q_D \uplus \{\mathsf{halt}\}) \cup (\mathcal{A} \uplus \{0\})$. If this is not the case, we can use several fields to achieve the required storage capacity. For $i \in \{1, 2\}$, let

$M_i \equiv$
  let *global* = new $\{\_ : I_1\}$ in
  let *aux* = new$\{_ : I_2\}$ in
  aux.tmp := new$\{_ : I_1\}$;
  new $\{$
    $\_ : I_5$;
    enq = $\lambda\_.$

```
assert (global.val ∈ Q_E);
let mine = new{_:I_1} in
let prev = aux.tmp in
  aux.tmp := mine;
  mine.val := π_1 δ(global.val);
  global.val := π_2 δ(global.val);
  new {
    _:I_3;
    step = λ_.
      assert (global.val ∈ Q_D);
      assert (prev.val = 0 and mine.val ≠ 0);
      global.state := δ(global.val, mine.val);
      mine.val := 0;
      if (aux.tmp = mine) then { global.val := halt; N_i }
      }
}
```

where $N_1 \equiv \mathsf{skip}$ and $N_2 \equiv \mathsf{div}$. □

Note that, if $\mathcal{Q}$ does not halt, M_1 and M_2 will generate the same complete plays and, otherwise, M_1 will have one more complete play corresponding to the last dequeuing operation. Consequently, $M_1 \cong M_2$ if and only $\mathcal{Q}$ does not halt. Hence, we have shown that $\cong$ is undecidable for iteration-free terms $\Delta|\emptyset \vdash M : I_5$.

Case 3

In this case we use the following interfaces.

$$\begin{aligned} I_1 &\equiv val : \mathsf{int} \\ I_2 &\equiv tmp : I \\ I_3 &\equiv step : \mathsf{void} \to \mathsf{void} \\ I_4 &\equiv run : I_3 \to \mathsf{void} \\ I_6 &\equiv enq : I_4 \to \mathsf{void} \end{aligned}$$

Note that $\mathcal{S}(\mathcal{S}(I_6)) = \{I_3\}$ and $I_3 \in \mathit{Intfs}_{\mathsf{m}}$. The undecidability argument will apply to interation-free terms of the form $\Delta|\emptyset \vdash M : I_6$.

In order to represent runs as plays over $\Delta, \emptyset \vdash I_6$, we need to change the representation scheme. We start off with $\star^{\emptyset}\, n^{\Sigma_0}$, where $\Sigma_0 = \{n \mapsto I_6\}$. Then each queue operation is represented as a two-move segment consisting of an O-move and a P-move.

- The ith enqueuing is represented by the segment

$$\mathsf{call}\, n.enq(o_i)^{\Sigma_i'} \quad \mathsf{call}\, o_i.run(n_i)^{\Sigma_i},$$

  where $\Sigma_i' = \Sigma_{i-1} \cup \{o_i : I_4\}$ and $\Sigma_i = \Sigma_i' \uplus \{n_i : I_3\}$. Note that we require that each enqueuing be associated with a fresh name n_i, but we do not need to impose an analogous condition on o_i.
- The ith dequeuing will be represented by

$$\mathsf{call}\, n_i.step(\star)^{\Sigma_j}\ \mathsf{ret}\, n_i.step(\star)^{\Sigma_j},$$

  where j is the number of enqueuings preceding the ith dequeuing.

Note that this time the moves corresponding to enqueuings are both questions. Consequently segments corresponding to sequences of queue operations, as specified above, will not be complete plays. However, they can easily be completed by adding j segments of the form $\mathsf{ret}\, o_i.run(\star)^{\Sigma_j}\, \mathsf{ret}\, n.enq(\star)^{\Sigma_j}$, where j is the number of enqueuings.

Given a queue machine $\mathcal{Q}$, let $R_{\mathcal{Q}}$ be the set containing representations of transition sequences of $\mathcal{Q}$. Define $R^+_{\mathcal{Q}}$ to be $R_{\mathcal{Q}}$ plus the completion of the play corresponding to the accepting run along with its prefixes, if such a run exists (i.e. if $\mathcal{Q}$ halts).

Lemma 32. *For any queue machine $\mathcal{Q}$, there exist $\Delta|\emptyset \vdash M_1, M_2 : I_6$ such that $[\![M_1]\!] = R_{\mathcal{Q}}$ and $[\![M_2]\!] = R^+_{\mathcal{Q}}$.*

Proof. Let $\mathcal{Q} = \{Q, Q_E, Q_D, \delta, q_I, \delta_E, \delta_D\}$. W.l.o.g. let us assume that int is large enough to accommodate $(Q_E \uplus Q_D \uplus \{\mathsf{halt}\}) \cup (\mathcal{A} \uplus \{0\})$. If this is not the case, we can use several fields to achieve the required storage capacity. Given $i \in \{1, 2\}$, let

$M_i \equiv$
```
  let global = new {_:I1} in
  let aux = new{ _:I2} in
  aux.tmp = new{ _:I1};
  new {
    _:I6;
    run = λx^I4.
        assert (global.val ∈ Q_E);
        let mine = new{_:I1} in
        let prev = aux.tmp in
          aux.tmp := mine;
          mine.val := π1δ(global.val);
          global.val := π2δ(global.val);
          x.run( new {
            _:I3;
           step = λ_.
               assert (global.val ∈ Q_D);
               assert (prev.val = 0 and mine.val ≠ 0);
               global.val := δ(global.val, mine.val);
               mine.val := 0;
               if (aux.tmp = mine) then global.state := halt
               });
           N_i
      }
```
where $N_1 \equiv \mathsf{div}$ and $N_2 \equiv \mathsf{assert}(\mathit{global.val} = \mathsf{halt})$. □

Note that if $\mathcal{Q}$ does not halt, M_1, M_2 will generate the same plays. If $\mathcal{Q}$ halts, M_2 will have one more complete play. Thus, $M_1 \cong M_2$ if and only $\mathcal{Q}$ does not halt. Consequently, $\cong$ is undecidable for iteration-free $\Delta|\emptyset \vdash M : I_6$.

D Canonical forms for IMJ* terms

To simplify the translation of IMJ* terms into automata, we rely on canonical forms.

Definition 33. We define IMJ* *canonical forms* $\mathcal{C}$ by:

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

We also forbid methods declared by $\mathsf{new}(\cdots)$ in a canonical form to call other methods of the same object. Moreover, in $\mathsf{let}\, x = \mathsf{new}\{\, y : \mathsf{B};\; \overrightarrow{m : \lambda \vec{z}.\mathcal{C}}\,\}\, \mathsf{in}\, \mathcal{C}'$ we require that $\mathcal{C}'$ does *not* contain subterms $v.\mathsf{m}(\vec{z})$, where v traces back to x via a chain of $\mathsf{let}\, v = (I)y\, \mathsf{in}\, \cdots$ constructs, where y is finally x.

Note that the restriction on internal calls of the same object does not affect expressivity: due to lack of recursion, such internal calls can be inlined. Also, our latter restriction does not render the methods of x uncallable ($\mathcal{C}'$ might return x^{B}).

Lemma 34. *Any* IMJ* *term* $\Delta|\Gamma \vdash M : \theta$ *can be effectively converted to an equivalent term* $\Delta|\Gamma \vdash \mathcal{C} : \theta$ *in canonical form.*

In the remainder of this section we prove the lemma above. We begin with an auxiliary result.

Lemma 35. $\mathsf{let}\, x = \mathcal{C}\, \mathsf{in}\, \mathcal{C}'$ *can be converted to an equivalent canonical form.*

Proof. Induction on the shape of $\mathcal{C}$. Recall that $\mathcal{C}$'s type is from B.

- $\mathsf{let}\, x = \mathsf{null}\, \mathsf{in}\, \mathcal{C}'$: Already in canonical form.
- $\mathsf{let}\, x = x^B\, \mathsf{in}\, \mathcal{C}'$: Take $\mathcal{C}'[x^B/x]$.
- $\mathsf{let}\, x = (I)\mathsf{new}\{\, t : B; \overrightarrow{m : \lambda \vec{y}.\mathcal{C}}\,\}\, \mathsf{in}\, \mathcal{C}'$: Take $\mathsf{let}\, y = \mathsf{new}\{\, t : B; \overrightarrow{m : \lambda \vec{y}.\mathcal{C}}\,\}\, \mathsf{in}\, \mathsf{let}\, x = (I)y\, \mathsf{in}\, \mathcal{C}'$. Consider $\mathcal{C}'$ and repeatedly replace any subterm $\mathsf{let}\, v = w.m(\vec{z})\, \mathsf{in}\, \mathcal{C}''$, where w refers to x (be it directly, i.e. $w \equiv x$, or through a chain of $\mathsf{let}\, z = (I)y\, \mathsf{in}\, \cdots$ bindings) either with a divergent term (if the relevant method is not available) or with $\mathsf{let}\, v = \mathcal{C}[\vec{z}/\vec{y}]\, \mathsf{in}\, \mathcal{C}''$. The rest follows by IH for $\mathcal{C}$.
- $$\begin{aligned}&\mathsf{let}\, x = (\mathsf{let}\, y = \mathcal{C}_{\mathsf{let}}\, \mathsf{in}\, \mathcal{C})\, \mathsf{in}\, \mathcal{C}' \cong \\ &\mathsf{let}\, y = \mathcal{C}_{\mathsf{let}}\, \mathsf{in}\, (\mathsf{let}\, x = \mathcal{C}\, \mathsf{in}\, \mathcal{C}')\end{aligned}$$
  and invoke IH for $\mathcal{C}$.
- Use
  $$\begin{aligned}&\mathsf{let}\, x = (\mathsf{if}\, y\, \mathsf{then}\, \mathcal{C}_1\, \mathsf{else}\, \mathcal{C}_2)\, \mathsf{in}\, \mathcal{C}' \cong \\ &\mathsf{if}\, y\, \mathsf{then}\, (\mathsf{let}\, x = \mathcal{C}_1\, \mathsf{in}\, \mathcal{C}')\, \mathsf{else}\, (\mathsf{let}\, x = \mathcal{C}_2\, \mathsf{in}\, \mathcal{C}')\end{aligned}$$
  and appeal to IH for $\mathcal{C}_1, \mathcal{C}_2$.

□

Proof (of Lemma 34). The main proof is by induction on term structure of M (whose type must be from R).

- (x^{B}): Already in canonical form.
- (null): Already in canonical form.

– (i): Take $\mathsf{let}\, x = i \,\mathsf{in}\, x^{\mathsf{int}}$.
– (a^{B}): Take $\mathsf{let}\, x = a \,\mathsf{in}\, x^{\mathsf{B}}$.
– (skip): Take $\mathsf{let}\, x = \mathsf{skip} \,\mathsf{in}\, x^{\mathsf{void}}$.
– $(M \oplus M')$: Take

$$\mathsf{let}\, x = \mathcal{C}_M \,\mathsf{in}\, (\mathsf{let}\, x' = \mathcal{C}_{M'} \,\mathsf{in}\, \mathsf{let}\, x'' = x \oplus x' \,\mathsf{in}\, x'')$$

and invoke Lemma 35 twice.
– $(M = M')$: Consider $\mathcal{C}_M = \mathcal{C}_{M'}$. Reason by induction on the shape of the canonical forms. The base cases are:
- $(I_1)\mathsf{new}(...) = (I_2)\mathsf{new}(...)$
- $(I)\mathsf{new}(...) = \mathsf{null}$
- $(I)\mathsf{new}(...) = x^B$
- $\mathsf{null} = \mathsf{null}$
- $\mathsf{null} = x^B$
- $x_1^B = x_2^B$

In case of casting errors, any divergent canonical form will do. Otherwise we obtain 0 in the first three cases, 1 in the fourth and the other cases can be reduced to

$$\mathsf{let}\, y = \mathsf{null} \,\mathsf{in}\, (\mathsf{let}\, z = (x^B = y) \,\mathsf{in}\, z).$$

and

$$\mathsf{let}\, z = (x_1 = x_2) \,\mathsf{in}\, z.$$

The let and if cases are dealt with using

$$(\mathsf{let}\, x = \mathcal{C}_1 \,\mathsf{in}\, \mathcal{C}_2) = M \cong \mathsf{let}\, x = \mathcal{C}_1 \,\mathsf{in}\, (\mathcal{C}_2 = M)$$
$$(\mathsf{if}\, x \,\mathsf{then}\, \mathcal{C}_1 \,\mathsf{else}\, \mathcal{C}_2) = M \cong \mathsf{if}\, x \,\mathsf{then}\, (\mathcal{C}_1 = M) \,\mathsf{else}\, (\mathcal{C}_2 = M)$$

and IH.
– $(x = x')$: Take $\mathsf{let}\, x'' = (x = x') \,\mathsf{in}\, x''$.
– $(M; M')$: Take $\mathsf{let}\, x = \mathcal{C}_M \,\mathsf{in}\, \mathcal{C}_{M'}$ and appeal to Lemma 35.
– $(\mathsf{if}\, M \,\mathsf{then}\, M' \,\mathsf{else}\, M'')$: Consider

$$\mathsf{let}\, x = \mathcal{C}_M \,\mathsf{in}\, (\mathsf{if}\, x \,\mathsf{then}\, \mathcal{C}_{M'} \,\mathsf{else}\, \mathcal{C}_{M''})$$

and appeal to Lemma 35.
– $(M.\mathsf{f} := M')$: Take $\mathsf{let}\, x = \mathcal{C}_M \,\mathsf{in}\, \mathsf{let}\, x' = \mathcal{C}_{M'} \,\mathsf{in}\, \mathsf{let}\, x'' = (x.\mathsf{f} := x') \,\mathsf{in}\, x''$ and appeal to Lemma 35.
– $(M.f)$: Induction on the shape of $\mathcal{C}_M$.
- (null): Then $\mathcal{C}_M.f$ causes an error (identified with non-termination).
- $((I)\mathsf{new}\{\, t \;:\; \mathsf{R} \;; \overline{\lambda \vec{x}.M}\,\})$: Then $\mathcal{C}_M.f$ is either a default value $v : \mathsf{B}$, i.e. $\mathsf{let}\, x = v \,\mathsf{in}\, x$ or error (identified with non-termination).
- (x^{B}): Take $\mathsf{let}\, x' = x.f \,\mathsf{in}\, x'$.
- $(\mathsf{let}\, x = \mathcal{C}_{\mathsf{let}} \,\mathsf{in}\, \mathcal{C})$: Use

$$(\mathsf{let}\, x = \mathcal{C}_{\mathsf{let}} \,\mathsf{in}\, \mathcal{C}).f \cong \mathsf{let}\, x = \mathcal{C}_{\mathsf{let}} \,\mathsf{in}\, (\mathcal{C}.f)$$

and invoke IH.

- $(\mathsf{if}\, x\, \mathsf{then}\, \mathcal{C}_1\, \mathsf{else}\, \mathcal{C}_2)$: Use

$$(\mathsf{if}\, x\, \mathsf{then}\, \mathcal{C}_1\, \mathsf{else}\, \mathcal{C}_2).f \cong \mathsf{if}\, x\, \mathsf{then}\, \mathcal{C}_1.f\, \mathsf{else}\, \mathcal{C}_2.f$$

  and invoke IH.

– $(y.f)$: Take $\mathsf{let}\, x = y.f\, \mathsf{in}\, x$.
– $(M.m(\vec{N}))$: By IH this is equivalent to $\mathcal{C}_M.m(\vec{\mathcal{C}_N})$. Now reason by induction on $\mathcal{C}_M$.
  - $(\mathsf{null}).m(\vec{\mathcal{C}_N})$: Take any divergent canonical form.
  - $(x^B).m(\vec{\mathcal{C}_N})$: Take

$$\mathsf{let}\, \vec{z} = \vec{\mathcal{C}_N}\, \mathsf{in}\, (\mathsf{let}\, v = x.m(\vec{z})\, \mathsf{in}\, v^G)$$

    and invoke Lemma 35.
  - $((I)\mathsf{new}\{\, t : \mathsf{R}\, ; \overline{m : \lambda\vec{x}.\mathcal{C}_M}\,\}.m(\vec{\mathcal{C}_N})$: If casting error or the object does not define a method corresponding to m, take any divergent canonical form. Otherwise consider $\mathsf{let}\, \vec{z} = \vec{\mathcal{C}_N}\, \mathsf{in}\, \mathcal{C}_M[\vec{z}/\vec{x}]$ and appeal to Lemma 35.
  - $(\mathsf{if}\, x\, \mathsf{then}\, \mathcal{C}_1\, \mathsf{else}\, \mathcal{C}_2).m(\vec{\mathcal{C}_N}) \cong \mathsf{if}\, x\, \mathsf{then}\, \mathcal{C}_1.m(\vec{\mathcal{C}_N})\, \mathsf{else}\, \mathcal{C}_2.m(\vec{\mathcal{C}_N})$
  - $(\mathsf{let}\, x = \mathcal{C}_{\mathsf{let}}\, \mathsf{in}\, \mathcal{C}).m(\vec{\mathcal{C}_N}) \cong \mathsf{let}\, x = \mathcal{C}_{\mathsf{let}}\, \mathsf{in}\, \mathcal{C}.m(\vec{\mathcal{C}_N})$
– $((I)M)$: Take $(I)\mathcal{C}_M$ and push (I) through if and let using

$$(I)\mathsf{if}\, x\, \mathsf{then}\, \mathcal{C}_1\, \mathsf{else}\, \mathcal{C}_2 \cong \mathsf{if}\, x\, \mathsf{then}\, (I)\mathcal{C}_1\, \mathsf{else}\, (I)\mathcal{C}_2$$
$$(I)\mathsf{let}\, x = \mathcal{C}_{\mathsf{let}}\, \mathsf{in}\, \mathcal{C} \cong \mathsf{let}\, x = \mathcal{C}_{\mathsf{let}}\, \mathsf{in}\, (I)\mathcal{C}$$

  Finally, replace $(I)\mathsf{null}$ with null, $(I)x^B$ with $\mathsf{let}\, y = (I)x\, \mathsf{in}\, y$ and $(I_1)(I_2)\mathsf{new}(\cdots)$ either with a divergent term (if casting error) or $(I_1)\mathsf{new}(\cdots)$.
– $(\mathsf{let}\, x = (I)y\, \mathsf{in}\, M)$: Take $\mathsf{let}\, x = (I)y\, \mathsf{in}\, \mathcal{C}_M$.
– $(\mathsf{let}\, x = y.m(\vec{N})\, \mathsf{in}\, M)$: Consider

$$\mathsf{let}\, \vec{z} = \vec{\mathcal{C}_N}\, \mathsf{in}\, (\mathsf{let}\, x = y.m(\vec{z})\, \mathsf{in}\, \mathcal{C}_M)$$

  and appeal to Lemma 35.
– $(\mathsf{let}\, x = M\, \mathsf{in}\, M')$: Appeal to Lemma 35.
– $(\mathsf{while}\, M\, \mathsf{do}\, M')$: Consider

$$\begin{array}{l}\mathsf{let}\, r = \mathsf{new}\{\, \_ : \mathsf{Var}_{\mathsf{int}};\, \}\, \mathsf{in}\\ \quad r.val := C_M; \mathsf{while}\, (r.val)\, \mathsf{do}\, (\mathcal{C}_{M'}; r.val := \mathcal{C}_M)\end{array}$$

  and follow with the same conversions as in the $M := M'$ and $M; M'$ cases.
– $(\mathsf{new}\{\, t : \mathsf{R}\, ; \overline{m : \lambda\vec{x}.M}\,\})$: Apply IH and take $\mathsf{new}\{\, t : \mathsf{R}\, ; \overline{m : \lambda\vec{x}.\mathcal{C}_M}\,\}$.

□

E Automata for IMJ*

In this section we produce the translation of canonical IMJ* terms into IMJ-automata. Let us start off by defining the semantics of IMJ-automata.

Definition 36. Let $\mathcal{A} = \langle Q, q_0, X_0, \delta, F\rangle$ be an IMJ-automaton and ρ_0 a register assignment with $\mathsf{dom}(\rho_0) = X_0$. We let the set of *configurations* of $\mathcal{A}$ be

$$\hat{Q} \;=\; Q \times \mathit{Reg} \times \mathit{Stk} \times \mathcal{P}_{\mathsf{fn}}(\mathbb{A})$$

and define ***configuration graph*** $(\hat{Q}, \rightarrow_\delta)$ of $(\mathcal{A}, \rho_0)$ as follows.

- For all $(q, \rho, \sigma, H) \in \hat{Q}$ and $q \xrightarrow{\nu X.(\mu^S, \phi)} q'$ in δ we have $(q, \rho, \sigma, H) \xrightarrow{m^\Sigma}_\delta (q', \rho', \sigma', H')$ when the following conditions are satisfied:
  - if $\mu^S \in \mathit{Lab}_{\mathsf{pop}}$ and $\phi = (s, Y, Z)$ then $\sigma = (s, \rho_s) :: \sigma'$ with $\mathsf{dom}(\rho_s) = Y$, $\mathsf{dom}(\rho) \cap \mathsf{dom}(\rho_s) = Z$ and $\rho \cup \rho_s$ a valid assignment, and $\rho_1 = \rho \cup \rho_s$; otherwise, $\rho_1 = \rho$;
  - if $X = \{\mathtt{r}_{i_1}, \cdots, \mathtt{r}_{i_m}\}$ then $\mathsf{dom}(\rho_1) \cap X = \emptyset$, $H' = H \cup \{a_1, \cdots, a_m\}$ and $\rho' = \rho_1[\mathtt{r}_{i_1} \mapsto a_1, \cdots, \mathtt{r}_{i_m} \mapsto a_m]$ where $a_1, \cdots, a_m$ are distinct names and:
    - if $q \in Q_O$ then $a_1, \cdots, a_m \notin \mathsf{cod}(\rho)$,
    - if $q \in Q_P$ then $a_1, \cdots, a_m \notin H$;
  - $m^\Sigma = \rho'(\mu^S)$;
  - if $\mu^S \in \mathit{Lab}_{\mathsf{push}}$ and $\phi = (s, Z)$ then $\sigma' = (s, \rho \restriction Z) :: \sigma$; otherwise, $\sigma' = \sigma$.
- For all $(q, \rho, \sigma, H) \in \hat{Q}$ and $q \xrightarrow{X} q'$ in δ we have $(q, \rho, \sigma, H) \xrightarrow{\epsilon}_\delta (q', \rho \restriction X, \sigma, H)$.
- For all $(q, \rho, \sigma, H) \in \hat{Q}$ and $q \xrightarrow{\pi} q'$ in δ we have $(q, \rho, \sigma, H) \xrightarrow{\epsilon}_\delta (q', \rho \circ \pi^{-1}, \sigma, H)$.

The set of strings *accepted* by $(\mathcal{A}, \rho_0)$ is defined as:

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

Note in particular that our machines accept on empty stack.

E.1 Translation of canonical terms

We next define the translation of canonical terms into IMJ-automata. The image of our translation will involve automata enjoying a strong discipline which implies determinacy. This will be shown in detail in Lemma 41. For now, we will require that each constructed automaton $\mathcal{A} = \langle Q, q_0, X_0, \delta, F\rangle$ satisfy the conditions:

1. for all $q \xrightarrow{\nu X.(\mu^S, \phi)} q'$, we have $\nu_{\mathtt{r}}(\mu^S, \phi) \subseteq \mathsf{dom}(S)$;
2. there is a *ranking function* $\mathsf{rank} : Q \to \{S \in \mathit{Sto}_{\mathtt{S}} \mid \nu_{\mathtt{r}}(S) \subseteq \mathsf{dom}(S)\}$ such that $\mathsf{dom}(\mathsf{rank}(q_0)) = X_0$.

Thus, we require that symbolic stores in transitions assign a symbolic value to all registers of the transition. The ranking function will assign to each state the unique symbolic store that state can be reached with. This will make the operation of the constructed automata easier to follow.

In the construction below we will be involving a notation for *combined transitions*, which are of the form:

$$q \xrightarrow{\nu X.(\Phi, \phi); Y} q' \;\equiv\; q \xrightarrow{\nu X.(\Phi, \phi)} q'' \xrightarrow{Y} q'$$

where the state q'' is chosen fresh and does not have other transitions. Such a state q'' will be called an ***intermediate state***. Moreover, for each permutation $\pi : \mathbb{C}_{\mathtt{r}} \xrightarrow{\cong} \mathbb{C}_{\mathtt{r}}$ we let $\pi^\dagger$ be the identity-extension of π to $Val_{\mathtt{s}}$. Finally, for any S, μ, we write $S@\mu$ for the subset of $\mathbb{C}_{\mathtt{r}}$ containing all $\mathtt{r}_i$'s recursively reachable in S from $\nu_{\mathtt{r}}(\mu)$.[5]

Let $\Delta|\Gamma \vdash \mathcal{C} : \theta$ be a term in canonical form with $\Gamma = \{x_1, \cdots, x_n\} \cup \{a_1 : I_1, \cdots, a_m : I_m\}$. Recall that 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:

$$(\!|\, \Delta|\Gamma \,|\!) = \{\mu_0^{S_0} \in Lab_{\mathtt{s}} \mid \exists \rho_0.\ \rho_0(\mu_0^{S_0}) \in P^1_{\Delta|\Gamma}\}$$

Thus, for Γ as above and $\mu_0^{S_0} \in (\!|\, \Delta|\Gamma \,|\!)$, μ_0 is of the form $(\ell_1, \cdots, \ell_n, \ell'_1, \cdots, \ell'_m)$, where each $\ell_i \in Val_{\mathtt{s}}$ and $\ell'_i \in \mathbb{C}_{\mathtt{r}}$, and $\mathsf{dom}(S_0) = S_0@\mu_0$. The canonical form will be translated into a family of automata indexed over $(\!|\, \Delta|\Gamma \,|\!)$. For each $\mu_0^{S_0}$, the corresponding automaton will accept, on initial assignment ρ_0, exactly the words w such that $\rho_0(\mu_0^{S_0})w$ is a representation of a complete play induced by the canonical form.

Given an IMJ* term $\Delta|\Gamma \vdash \mathcal{C} : \theta$ in canonical form we define a family of automata $(\!|\, \mathcal{C} \,|\!) = \{\, (\!|\, \mathcal{C} \,|\!)_\Phi \mid \Phi \in (\!|\, \Delta|\Gamma \,|\!) \,\}$ by induction on the shape of $\mathcal{C}$. The precise number of registers (i.e. the value of R) will be left implicit and can be calculated by reference to the constituent automata. Let us assume that $\Phi = \mu_0^{S_0}$, take $X_0 = \mathsf{dom}(S_0)$ and let $Z_0 \subseteq \mathbb{C}_{\mathtt{r}}$ contain all the registers which appear in μ_0. Our automata will obey the invariant that the registers in their respective Z_0 are never deleted or overwritten. The base and inductive cases are as follows. In all cases we stipulate $\mathsf{rank}(q_0) = S_0$.

- $(\!|\, \mathsf{null} \,|\!)_\Phi = \ q_0 \xrightarrow{\mathtt{nul}^{S_0}} q_F$, with $\mathsf{rank}(q_F) = S_0$.
- $(\!|\, x \,|\!)_\Phi = q_0 \xrightarrow{\ell_k^{S_0}} q_F$, where $x \equiv x_k$ and $\mathsf{rank}(q_F) = S_0$.
- $(\!|\, \mathsf{let}\, y = \Xi\, \mathsf{in}\, \mathcal{C} \,|\!)_\Phi = (\!|\, \mathcal{C} \,|\!)_{\mu_0'^{S_0}}$, where $\mu'_0 = \mu_0\ell$ is the extension of μ_0 with the symbolic value ℓ, which is: - nul, if $\Xi \equiv \mathsf{null}$; - i, if $\Xi \equiv i$; - $\star$, if $\Xi \equiv \mathsf{skip}$.
- $(\!|\, \mathsf{if}\, x\, \mathsf{then}\, \mathcal{C}_1\, \mathsf{else}\, \mathcal{C}_0 \,|\!)_\Phi = (\!|\, \mathcal{C}_j \,|\!)_\Phi$, where $x \equiv x_k$ and $j = 0$ iff $\ell_k = 0$.
- $(\!|\, \mathsf{let}\, z = (x.\mathsf{f} := y)\, \mathsf{in}\, \mathcal{C} \,|\!)_\Phi = q_0 \xrightarrow{X} (\!|\, \mathcal{C} \,|\!)_{\mu_0 \star^{S'_0}}$, where $x \equiv x_k$, $\ell_k = \mathtt{r}_{k'}$, $y \equiv x_j$, $\ell_j = \mathtt{r}_{j'}$ and $S' = S_0[\mathtt{r}_{k'} \mapsto (S_0(\mathtt{r}_{k'})[\mathsf{f} \mapsto \mathtt{r}_{j'}])]$, and the initial transition empties all the registers which cannot be reached from μ_0 via S', that is, $X = S'@\mu_0$. Moreover, $S'_0 = S' \restriction X$.
- For $\mathcal{A} = (\!|\, \mathsf{let}\, y = (x_1 = x_2)\, \mathsf{in}\, \mathcal{C} \,|\!)_\Phi$, with $x_i \equiv x_{k_i}$, $\ell_{k_i} = \mathtt{r}_{k'_i}$, for $i = 1, 2$, we let $\mathcal{A} = (\!|\, \mathcal{C} \,|\!)_{\mu_0 j^{S_0}}$, where $j = 1$ if $k'_1 = k'_2$ (otherwise, $j = 0$).
- For $\mathcal{A} = (\!|\, \mathsf{let}\, y = (I)x\, \mathsf{in}\, \mathcal{C} \,|\!)_\Phi$, with $x \equiv x_k$ and $\ell_k = \mathtt{r}_{k'}$, we let $\mathcal{A} = (\!|\, \mathcal{C} \,|\!)_{\mu_0 \mathtt{r}_{k'}^{S_0}}$ if $\Delta \vdash S_0(\mathtt{r}_{k'}) \leq I$; otherwise, $\mathcal{A}$ is empty.
- $(\!|\, \mathsf{let}\, y = x.\mathsf{f}\, \mathsf{in}\, \mathcal{C} \,|\!)_\Phi = (\!|\, \mathcal{C} \,|\!)_{\mu_0'^{S_0}}$, where $x \equiv x_k$, $\ell_k = \mathtt{r}_{k'}$ and $\mu'_0 = \mu_0\ell$ is the extension of μ_0 with the symbolic value $\ell = S_0(\mathtt{r}_{k'}).\mathsf{f}$.
- $(\!|\, \mathsf{let}\, x^{\mathsf{int}} = y.\mathsf{m}(\vec{z})\, \mathsf{in}\, \mathcal{C} \,|\!)_\Phi =$

$$q_0 \xrightarrow{\mathsf{call}\, \mathtt{r}_j.\mathsf{m}(\vec{\ell'})^{S_0}} q_1 \xrightarrow{\nu X.\mathsf{ret}\, \mathtt{r}_j.\mathsf{m}(j')^{S'_0}; Y} (\!|\, \mathcal{C} \,|\!)_{\Phi'}$$

[5] That is, $S@\mu = S^*(\nu_{\mathtt{r}}(\mu))$ where, for each $X \subseteq \mathbb{C}_{\mathtt{r}}$, $S^*(X) = \bigcup_{i \in \omega} S^i(X)$ with $S^0(X) = X$ and $S^{i+1}(X) = S^i(X) \cup \nu_{\mathtt{r}}(S \restriction S^i(X)))$.

where $y \equiv x_k$, $\ell_k = \mathtt{r}_j$, $\Phi' = \mu_0 j'^{S_0'}$, j' ranges over $[0, \textsc{MaxInt}]$ and $\vec{\ell'}$ are the symbolic counterparts of $\vec{z}$ (as determined by Γ, μ_0). Moreover, S_0' ranges over all symbolic stores such that $\mu_0 j'^{S_0'} \in (\!| \Delta | \Gamma, y : \mathsf{int} |\!)$, while $X = \mathsf{dom}(S_0') \setminus X_0$ and $Y = \mathsf{dom}(S_0')$. The selection of X, S_0' adheres to the proviso that we select one representative transition $\nu X.\mathsf{ret}\,\mathtt{r}_j.\mathsf{m}(j')^{S_0'}$ from each equivalence class induced from treating ν as a binder. Finally, $\mathsf{rank}(q_1) = S_0$ and the rank of each intermediate state above is S_0'.

– $(\!|\, \mathsf{let}\, x^I = y.\mathsf{m}(\vec{z})\, \mathsf{in}\, \mathcal{C} \,|\!)_\Phi =$

$$q_0 \xrightarrow{\mathsf{call}\,\mathtt{r}_j.\mathsf{m}(\vec{\ell})^{S_0}} q_1 \xrightarrow{\nu X.\mathsf{ret}\,\mathtt{r}_j.\mathsf{m}(\mathtt{r}_{j'})^{S_0'}\,;\,Y} (\!|\, \mathcal{C} \,|\!)_{\Phi'}$$

where $y \equiv x_k$, $\ell_k = \mathtt{r}_j$, $\Phi' = \mu_0 \mathtt{r}_{j'}^{S_0'}$ and $\vec{\ell'}$ are the symbolic counterparts of $\vec{z}$ (as determined by Γ, μ_0). Moreover, j', S_0' range over all indices and symbolic stores respectively such that $\mu_0 \mathtt{r}_{j'}^{S_0'} \in (\!| \Delta | \Gamma, y : \mathsf{int} |\!)$. X, Y and rank are selected as in the previous case.

– For $\mathcal{A} = (\!|\, \mathsf{let}\, z = (\mathsf{while}\, r.val\, \mathsf{do}\, \mathcal{C}_1)\, \mathsf{in}\, \mathcal{C}_2 \,|\!)_\Phi$, it is useful to first define a construction on IMJ-automata which will allow us to abstract away registers and their symbolic values. Given an IMJ-automaton $\mathcal{A} = \langle Q, q_0, X_0, \delta, F \rangle$ with ranking function rank and a symbolic store R_0 such that $R_0 \subseteq \mathsf{rank}(q_0)$, we define the IMJ-automaton $\nu R_0.\mathcal{A} = \langle Q', q_0', X_0', \delta', F' \rangle$ by setting:

$$Q' = \{(q, R) \mid R \subseteq \mathsf{rank}(q) \wedge \mathsf{dom}(R) \cap \mathsf{cod}(\mathsf{rank}(q) \setminus R) = \emptyset\}$$

$q_0' = (q_0, R_0)$, $X_0' = X_0 \setminus \mathsf{dom}(R_0)$, $F' = \{(q, R) \in Q' \mid q \in F\}$ and we take as new ranking function:

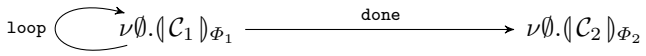
$$\mathsf{rank}' = \{(q, R) \mapsto \mathsf{rank}(q) \setminus R\}$$

where note that $\nu_{\mathtt{r}}(\mathsf{rank}(q) \setminus R) \subseteq \mathsf{dom}(\mathsf{rank}(q) \setminus R)$ is ensured by $\mathsf{cod}(\mathsf{rank}(q) \setminus R) \cap \mathsf{dom}(R) = \emptyset$. Finally, δ' is constructed by including precisely the following transitions, for each q_O, q_P and R such that $(q_O, R), (q_P, R) \in Q'$.

- For each $q_O \xrightarrow{X} q_O'$ in $\mathcal{A}$ with $\mathsf{dom}(R) \subseteq X$, add a transition $(q_O, R) \xrightarrow{X'} (q_O', R)$ where $X' = X \setminus \mathsf{dom}(R)$.
- For each $q_P \xrightarrow{X} q_P'$ in $\mathcal{A}$, add a transition $(q_P, R) \xrightarrow{X'} (q_P', R')$ with $R' = R \restriction X$ and $X' = X \setminus \mathsf{dom}(R)$.
- For each $q_O \xrightarrow{\pi} q_O'$ in $\mathcal{A}$, we add a transition $(q_O, R) \xrightarrow{\pi} (q_O', \pi^\dagger \circ R \circ \pi^{-1})$.
- For each $q_O \xrightarrow{\nu X.(\mu^S, \phi)} q_P'$ in $\mathcal{A}$ and each $R \subseteq S$ such that $\mathsf{dom}(R) \cap (X \cup \mathsf{cod}(S \setminus R) \cup \nu_{\mathtt{r}}(\mu)) = \emptyset$, we add a transition $(q_O, R) \xrightarrow{\nu X.(\mu^{S'}, \phi')} (q_P', R)$ where:
  - $S' = S \setminus R$,
  - $\phi' = ((q, R), Y \setminus \mathsf{dom}(R), Z \setminus \mathsf{dom}(R))$ if $\mu^S \in Lab_{\mathsf{pop}}$ and $\phi = (q, Y, Z)$, otherwise $\phi' = \phi$.
- For each $q_P \xrightarrow{\nu X.(\mu^S, \phi)} q_O'$ in $\mathcal{A}$, add a transition $(q_P, R) \xrightarrow{\nu X'.(\mu^{S'}, \phi')} (q_O', R')$ where:

$*$ $S' = S \restriction Z$, $R' = S \setminus S'$ and $X' = (X \cup \mathsf{dom}(R)) \cap Z$,
$*$ $Z = (S@\mu) \cup S^*(\mathsf{dom}(S) \setminus (X \cup \mathsf{dom}(R)))$,
$*$ $\phi' = \phi$ if $\mu^S \in Lab_{\mathtt{noop}}$, otherwise if $\phi = (q, Y)$ then $\phi' = ((q, R'), Y \setminus \mathsf{dom}(R'))$.

Back to $\mathcal{A}$, suppose $r \equiv x_k$ and $\ell_k = \mathtt{r}_{k'}$. If $S_0(\mathtt{r}_{k'}).val = 0$ then $\mathcal{A} = (\!|\, \mathcal{C}_2 \,|\!)_\Phi$. Otherwise, we combine a family of modified copies of $\nu\emptyset.(\!|\, \mathcal{C}_1 \,|\!)_{\Phi_1}$ and a family of $\nu\emptyset.(\!|\, \mathcal{C}_2 \,|\!)_{\Phi_2}$, containing one copy each for each $\Phi_1 = \mu_0^{S_1}$ and $\Phi_2 = \mu_0 \star^{S_2}$ which range over $(\!|\, \Delta | \Gamma \,|\!)$ (but μ_0 is fixed). We construct $\mathcal{A}$ as follows.



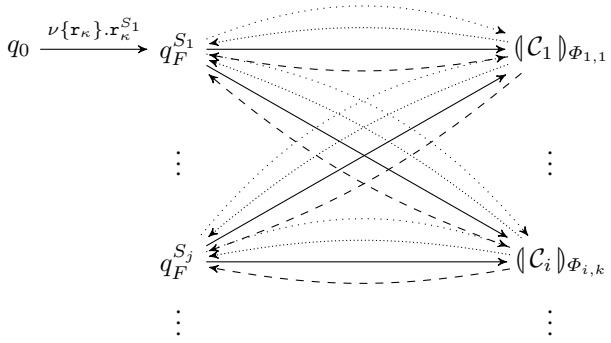
The initial state of $\mathcal{A}$ is the initial state of $\nu\emptyset.(\!|\, \mathcal{C}_1 \,|\!)_\Phi$. The families of transitions `loop` and `done` are explained below.

- `loop`. We replace each transition $(q, R) \xrightarrow{\nu X. \star^{S'}} (q_F, R')$ in $\nu\emptyset.(\!|\, \mathcal{C}_1 \,|\!)_{\Phi_1}$, where q_F final and $S'(\mathtt{r}_{k'}) \neq 0$, with an ϵ-transition labelled $\mathsf{dom}(S'_1) \backslash \mathsf{dom}(R'')$ from (q, R) to state (q_1, R'') of $\nu\emptyset.(\!|\, \mathcal{C}_1 \,|\!)_{\Phi'_1}$, where q_1 the initial state of $(\!|\, \mathcal{C}_1 \,|\!)_{\Phi'_1}$, $S'_1 = (S' \cup R') \restriction Z_1$, $Z_1 = (S' \cup R')@\mu_0$, $R'' = S'_1 \restriction Z'_1$, $Z'_1 = (\mathsf{dom}(R) \cup X) \cap Z_1$ and $\Phi'_1 = \mu_0^{S'_1}$.
- `done`. We replace each transition $(q, R) \xrightarrow{\nu X. \star^{S'}} (q_F, R')$ in $\nu\emptyset.(\!|\, \mathcal{C}_1 \,|\!)_{\Phi_1}$, where q_F final and $S'(\mathtt{r}_{k'}) = 0$, with an ϵ-transition from (q, R) to the state (q_2, R'') of $\nu\emptyset.(\!|\, \mathcal{C}_2 \,|\!)_{\Phi_2}$, where q_2 initial in $(\!|\, \mathcal{C}_2 \,|\!)_{\Phi_2}$, similarly as above.

– For $\mathcal{A} = (\!|\, (I')\mathsf{new}\{\, x : I; \mathsf{m}_1 : \lambda\vec{x}_1.\mathcal{C}_1, \cdots, \mathsf{m}_n : \lambda\vec{x}_n.\mathcal{C}_n \,\}\,|\!)_\Phi$, if $\Delta \not\vdash I \leq I'$ then $\mathcal{A}$ is empty. Otherwise, we construct $\mathcal{A}$ as follows. Let $\mathtt{r}_\kappa$ be the first register not in X_0 and set $Z'_0 = Z_0 \cup \{\mathtt{r}_\kappa\}$. Moreover, let $S_1, \cdots, S_\lambda$ be an enumeration of all symbolic stores S such that $\mu_0 \mathtt{r}_\kappa^S \in (\!|\, \Delta | \Gamma, x : I \,|\!)$ and S_1 be the extension of S_0 which assigns default (symbolic) values to $\mathtt{r}_\kappa$. Moreover, for each $i \in [1, n]$, let $(\vec{\ell}_1, S'_1), \cdots, (\vec{\ell}_{\lambda_i}, S'_{\lambda_i})$ be an enumeration of all sequences of symbolic values $\vec{\ell}$ and symbolic stores S' such that $\mu_0 \mathtt{r}_\kappa \vec{\ell}^{S'} \in (\!|\, \Delta | \Gamma, x : I, \overrightarrow{x_i : \theta_i} \,|\!)$. We define $\mathcal{A}$ as an automaton which combines states $q_0, q_F^{S_j}$ $(j = 1, \cdots, \lambda)$ and a family of modified copies of $(\!|\, \mathcal{C}_i \,|\!)_{\Phi_{i,k}}$, for each $k = 1, \cdots, \lambda_i$ and $\Phi_{i,k} = \mu_0 \mathtt{r}_\kappa \vec{\ell}_k^{S'_k}$ as below.

- We connect q_0 to $q_F^{S_1}$ with a transition labelled $\nu\{\mathtt{r}_\kappa\}.\mathtt{r}_\kappa^{S_1}$ which introduces a fresh name and stores it in register κ.
- For each $i \in [1, n]$ and $k \in [1, \lambda_i]$ we add a copy of $(\!|\, \mathcal{C}_i \,|\!)_{\Phi_{i,k}}$.

Each such $(\!|\, \mathcal{C}_i \,|\!)_{\Phi_{i,k}}$ produces plays corresponding to calls of $\mathcal{C}_i$ on input that is symbolically represented by $\vec{\ell}_k^{S'_k}$. The resulting automaton $\mathcal{A}$ looks as follows.



For each j, we set $\mathsf{rank}(q_F^{S_j}) = S_j$. The transitions between the $q_F^{S_j}$'s and the $(\!|\mathcal{C}_i|\!)_{\Phi_{i,k}}$'s are defined below.

- ($\longrightarrow$) We connect each $q_F^{S_j}$ to the initial state of $(\!|\mathcal{C}_i|\!)_{\Phi_{i,k}}$ with a combined transition labelled $\nu X.\mathsf{call}\,\mathtt{r}_\kappa.\mathsf{m}_i(\vec{\ell}_k)^{S'_k}; Y$, where $Y = \mathsf{dom}(S'_k)$ and $X = Y \setminus \mathsf{dom}(S_j)$, making sure we do not include duplicate transitions (modulo α-equivalence induced by ν). These transitions correspond to calls of $\mathcal{C}_i$.
- ($\dashrightarrow$) We replace each transition $q \xrightarrow{\nu X.\ell^S} q_F$ of $(\!|\mathcal{C}_i|\!)_{\Phi_{i,k}}$, with q_F a final state and S such that $S \restriction Z = S_j$ and $Z = S@(\mu_0\mathtt{r}_\kappa)$, with a combined transition from q to $q_F^{S_j}$ with label $\mathsf{ret}\,\mathtt{r}_\kappa.\mathsf{m}_i(\ell)^S; Z$. These transitions correspond to returns of $\mathcal{C}_i$.
- ($\dashrightarrow\!\!\!\!\!\!\cdots$) For each $\iota \neq \kappa$, method m and P-state q we replace each transition sequence $q \xrightarrow{\nu X.\mathsf{call}\,\mathtt{r}_\iota.\mathsf{m}(\vec{\ell})^S} q' \xrightarrow{\nu X'.\mathsf{ret}\,\mathtt{r}_\iota.\mathsf{m}(\ell')^{S'}} q''$ in $(\!|\mathcal{C}_i|\!)_{\Phi_{i,k}}$ with the following transitions. Taking $Z = S@(\mu_0\mathtt{r}_\kappa)$, $S_j = S \restriction Z$ and $Y = \mathsf{dom}(S)$, we add a combined transition from q to $q_F^{S_j}$ labelled

$$\nu X.\mathsf{call}\,\mathtt{r}_\iota.\mathsf{m}(\vec{\ell})^S/(q'', Y)\ ; Z$$

  and rank the intermediate state with S. Moreover, for each $q_F^{S_{j'}}$ and partial bijection $f : (\mathsf{dom}(S_{j'}) \setminus Z'_0) \xrightarrow{\cong} X' \cup (Y \setminus Z'_0)$, we add a sequence of transitions (for a freshly introduced middle state with rank $\pi^\dagger \circ S_{j'} \circ \pi^{-1}$):

$$q_F^{S_{j'}} \xrightarrow{\pi} \xrightarrow{\nu X''.\mathsf{ret}\,\mathtt{r}_\iota.\mathsf{m}(\ell')^{S'},(q'',Y,Y')} q''$$

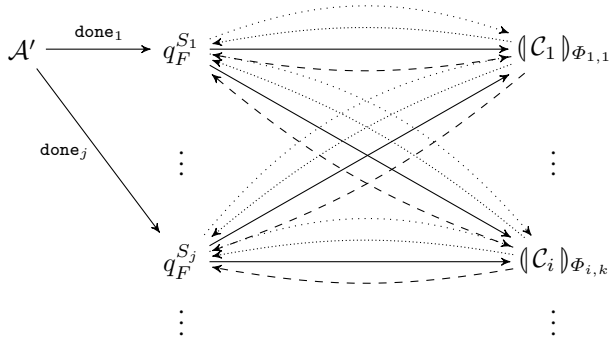
  where $X'' = X' \setminus \mathsf{cod}(f)$, $Y' = \mathsf{cod}(f) \cap Y$ and $\pi : \mathbb{C}_\mathtt{r} \xrightarrow{\cong} \mathbb{C}_\mathtt{r}$ is some unique extension of f satisfying:
  - $\pi(r) = r$ for all $r \in Z'_0$;
  - $\pi(\mathsf{dom}(S_{j'})) \cap (X' \cup Y) \subseteq (X' \setminus X'') \cup Y'$.

  These transitions correspond to calls/returns to methods of objects in $\mu_0\vec{\ell}_k^{S'_k}$.

Note that the construction obeys the invariant that, for each $q_F^{S_j}$ and reachable configuration $(q_F^{S_j}, \rho, \sigma, H)$, $\mathsf{dom}(\rho) = \mathsf{dom}(S_j)$.

– For $\mathcal{A} = (\!|\, \mathsf{let}\, x = \mathsf{new}\{\, x : I;\ \overrightarrow{\mathsf{m} : \lambda\vec{y}.\mathcal{C}}\,\}\,\mathsf{in}\,\mathcal{C}'\,|\!)_{\Phi}$, with $I \in \mathsf{B}$, let $\mathtt{r}_\kappa$ be the first register in $\mathbb{C}_{\mathtt{r}} \setminus X_0$ and take $Z_0' = Z_0 \cup \{\mathtt{r}_\kappa\}$. We first construct $\mathcal{A}' = \nu R_0.(\!|\mathcal{C}'|\!)_{\mu_0 \mathtt{r}_\kappa^{S_0 \cup R_0}}$ where $R_0 = \{\mathtt{r}_\kappa \mapsto (I, \Psi_0)\}$ and Ψ_0 contains the default symbolic values for I. We observe that, in $\mathcal{A}'$,
  - because of the restrictions on $\mathcal{C}'$, there are no transitions involving calls or returns of methods from $\mathtt{r}_\kappa$;
  - on the other hand, $\mathcal{A}'$ might be returning $\mathtt{r}_\kappa$ as its final value.

  These properties allow us to construct $\mathcal{A}$ by trailing $\mathcal{A}'$ with the automaton for $\mathsf{new}\{\, x : I;\ \overrightarrow{\mathsf{m} : \lambda\vec{y}.\mathcal{C}}\,\}$ from the previous case, as below.



  The initial state of $\mathcal{A}$ is that of $\mathcal{A}'$, while the final states are the final states of $\mathcal{A}'$ and q_F. It remains to specify $\mathtt{done}_j$: we replace each transition $q_P \xrightarrow{\nu X.\mathtt{r}_\kappa^S} q_F'$ of $\mathcal{A}'$, where q_F' final, $S_j = S \restriction Z$ and $Z = S@(\mu_0 \mathtt{r}_\kappa)$, with a combined transition from q_P to $q_F^{S_j}$ with label $\nu X.\mathtt{r}_\kappa^S\,; Z$.

The construction above replicates the (inductive) game-semantic interpretation [11] of canonical terms, apart from the stores that appear in moves. For the latter, the translation uses representations according to the names that P can actually use at each point in the game. The concrete plays are then retrieved from such representations via the extension operation, which dictates that names that P does not use can have their values changed only by O (that is, P must always copycat their values).

E.2 Determinacy and canonicity of IMJ-automata

We next examine static properties pertaining to determinacy of IMJ-automata and focus on a particular class of automata which are canonical in a very strong sense. We start off by giving a formal definition of determinacy.

Definition 37. We say that an IMJ-automaton $\mathcal{A}$ is ***deterministic*** if, for any compatible[6] ρ_0 and reachable configuration $\hat{q}$ of $(\mathcal{A}, \rho_0)$ and any label $x \in \mathbb{W}$, there is at most one way to accept x from $\hat{q}$. That is, if $\hat{q} \xrightarrow{\epsilon} \!\!\!\rightarrow_\delta \xrightarrow{x}_\delta \hat{q}_1$ and $\hat{q} \xrightarrow{\epsilon} \!\!\!\rightarrow_\delta \xrightarrow{x}_\delta \hat{q}_2$ then $\hat{q}_1 = \hat{q}_2$.

Determinacy is a dynamic notion, depending on the reachable configurations of an automaton. We introduce a static notion which ensures determinacy.

[6] i.e. such that $\mathsf{dom}(\rho_0) = X_0$, where $\mathcal{A} = \langle \cdots, X_0, \cdots \rangle$.

Definition 38. We say that an IMJ-automaton $\mathcal{A} = \langle Q, q_0, X_0, \delta, F\rangle$ is ***strongly deterministic*** if, for any $q \in Q$:

- if $q \in Q_P$ then $|\delta \upharpoonright \{q\}| \leq 1$, i.e. there is at most one transition out of q;
- if $q \in Q_O$ and there is some $q \xrightarrow{X} q'$ in δ then $|\delta \upharpoonright \{q\}| = 1$;
- if $q \in Q_O$, $\Phi \in Lab_{\mathsf{pop}}$ and there is some $q \xrightarrow{\nu X.\Phi,\phi} q'$ in δ then $|\delta \upharpoonright \{q\}| = 1$;
- if $q \in Q_O$ and $q \xrightarrow{\pi} q'$ in δ then there is some $q' \xrightarrow{\nu X.\Phi,\phi} q''$ in δ with $\Phi \in Lab_{\mathsf{pop}}$;
- if $q \in Q_O$, $\Phi_i \in Lab_{\mathsf{pop}}$, $\phi_i = (s_i, Y_i, Z_i)$ and $q \xrightarrow{\pi_i} \xrightarrow{\nu X_i.\Phi_i,\phi_i} q_i$, for $i = 1, 2$, then $\phi_1 = \phi_2$, $Z_1' = Z_2'$ and $\nu(X_1' \cup \Xi_1').\Phi_1' \sim_\alpha \nu(X_2' \cup \Xi_2').\Phi_2'$ imply $(\pi_1, X_1, \Phi_1, q_1) = (\pi_2, X_2, \Phi_2, q_2)$, where $\Xi_i = Y_i \setminus Z_i$ and $(X_i', \Phi_i', Z_i', \Xi_i') = \pi_i^{-1} \cdot (X_i, \Phi_i, Z_i, \Xi_i)$;
- if $q \in Q_O$, $\Phi_i \in Lab_{\mathsf{noop}}$ and $q \xrightarrow{\nu X_i.\Phi_i} q_i$, for $i = 1, 2$, then $\nu X_1.\Phi_1 \sim_\alpha \nu X_2.\Phi_2$ implies $(X_1, \Phi_1, q_1) = (X_2, \Phi_2, q_2)$.

Above, we write $\sim_\alpha$ for the alpha-equivalence relation induced by treating ν as a binder.[7] Moreover, for any symbolic entity y and permutation π of $\mathbb{C}_{\mathtt{r}}$, $\pi \cdot y$ is obtained by recursively applying π to all registers inside y.

Lemma 39. *If $\mathcal{A}$ is strongly deterministic then it is deterministic.*

Proof. Let $\hat{q} = (q, \rho, \sigma, H)$ be any configuration of $\mathcal{A}$. It suffices to consider the cases where δ can have more than one transition out of q. That is, we can assume WLOG that $q \in Q_O$ and that there are no pop or set transitions out of q. Thus, q has only noop and permutations as outgoing transitions. Since each permutation must necessarily be followed by a pop, and pop and noop labels cannot be made equal, it suffices to consider the two transition classes separately. So, suppose q has two outgoing transitions $q \xrightarrow{\nu X_i.\Phi_i} q_i$ with $\Phi_i \in Lab_{\mathsf{noop}}$, for $i = 1, 2$, which produce the same label $x \in \mathbb{W}$. Then, $x = \rho[\vec{r}_1 \mapsto \vec{a}_1](\Phi_1) = \rho[\vec{r}_2 \mapsto \vec{a}_2](\Phi_2)$, where $\vec{r}_i$ an enumeration of X_i and $\vec{a}_i$ locally fresh names. Since all the registers in X_i appear in Φ_i and $\rho[\vec{r}_1 \mapsto \vec{a}_1](\Phi_1) = \rho[\vec{r}_2 \mapsto \vec{a}_2](\Phi_2)$, it must be the case that $\vec{a}_1$ is a permutation of $\vec{a}_2$. Let now π be some permutation such that:

- if $j \in X_2$ and $a_{1j'} = a_{2j}$ then $\pi(j) = j'$;
- if $j \notin X_1 \cup X_2$ then $\pi(j) = j$.

We can now observe that $(X_1, \Phi_1) = \pi \cdot (X_2, \Phi_2)$. Finally, if $q \xrightarrow{\pi_i} \xrightarrow{\nu X_i.\Phi_i,\phi_i} q_i$ with $\Phi_i \in Lab_{\mathsf{pop}}$, for $i = 1, 2$, produce the same label $x \in \mathbb{W}$ then it must be the case that ϕ_1 and ϕ_2 are equal to some (s, Y, Z), where $\sigma = (s, \rho') :: \sigma'$ and $Y = \mathsf{dom}(\rho')$ and $Z = \{\mathtt{r}_k \mid \rho'(\mathtt{r}_k) \in \mathsf{cod}(\rho)\}$. We must have that $(\rho \circ \pi_1^{-1})(Z) = (\rho \circ \pi_2^{-1})(Z)$, which implies $\rho(\pi_1^{-1}(Z)) = \rho(\pi_2^{-1}(Z))$ and therefore, since ρ must be defined in $\pi_i^{-1}(Z)$, we have $\pi_1^{-1} \cdot Z = \pi_2^{-1} \cdot Z$. Moreover, production of the same label x means that

$$(\rho \circ \pi_1^{-1})[\vec{r} \mapsto \vec{b}][\vec{r}_1 \mapsto \vec{a}_1](\Phi_1) = (\rho \circ \pi_2^{-1})[\vec{r} \mapsto \vec{b}][\vec{r}_2 \mapsto \vec{a}_2](\Phi_2)$$

[7] i.e. $\nu X.x \sim_\alpha \nu Y.y$ if there is a permutation π of $\mathbb{C}_{\mathtt{r}}$ such that $(X, x) = \pi \cdot (Y, y)$ and $\pi(r) = r$ for all $r \notin X \cup Y$.

where $\vec{r}$ is an enumeration of the set $\Xi = Y \setminus Z$ and $\vec{r}_i$ an enumeration of X_i (for $i = 1, 2$). The above can be rewritten as:

$$\rho[(\pi_1^{-1} \cdot (\vec{r}, \vec{r}_1)) \mapsto (\vec{b}, \vec{a}_1)](\Phi_1') = \rho[(\pi_2^{-1} \cdot (\vec{r}, \vec{r}_2)) \mapsto (\vec{b}, \vec{a}_2)](\Phi_2')$$

and thus, working as above, we obtain $\nu(X_1' \cup \Xi_1').\Phi_1' \sim_\alpha \nu(X_2' \cup \Xi_2').\Phi_2'$. □

The automata we obtain from canonical forms via our recursive construction obey a stronger discipline than strong determinacy. Going over our construction we can verify the following constraints.

Definition 40. Let $\mathcal{A} = \langle Q, q_0, X_0, \delta, F \rangle$ be an IMJ-automaton. We say that $\mathcal{A}$ is ***canonical*** if it is strongly deterministic and, in addition:

1. whenever $q_O \xrightarrow{\nu X.(\Phi,\phi)} q_P$ in δ, with $q_O \in Q_O$, q_P is an intermediate state;
2. for all $q \xrightarrow{\nu X.(\mu^S,\phi)} q'$ we have $\nu_{\mathtt{r}}(\mu^S, \phi) \subseteq \mathsf{dom}(S)$;
3. there is a *ranking function*

$$\mathsf{rank} : Q \to \{ S \in Sto_{\mathtt{S}} \mid \nu_{\mathtt{r}}(S) \subseteq \mathsf{dom}(S) \}$$

such that $\mathsf{dom}(\mathsf{rank}(q_0)) = X_0$ and, for any $q \in Q$:
   - if $q \xrightarrow{X} q'$ in δ then $X \subseteq \mathsf{dom}(\mathsf{rank}(q))$ and $\mathsf{rank}(q') = \mathsf{rank}(q) \upharpoonright X$;
   - if $q \xrightarrow{\pi} q'$ in δ then $\mathsf{rank}(q') = \pi^\dagger \circ \mathsf{rank}(q) \circ \pi^{-1}$;
   - if $q \xrightarrow{\nu X.(\mu^S,\phi)} q'$ in δ then $\mathsf{dom}(\mathsf{rank}(q)) \cap X = \emptyset$, $\mathsf{rank}(q') = S$ and if $q \in Q_P$ then $\mathsf{dom}(\mathsf{rank}(q)) \subseteq \mathsf{dom}(S)$.

In particular, if $\mathcal{A}$ is canonical then all the symbolic stores that appear in its transitions are closed (i.e. satisfy $\nu_{\mathtt{r}}(S) \subseteq \mathsf{dom}(S)$).

Lemma 41. *Given an* IMJ* *term* $\Delta|\Gamma \vdash \mathcal{C} : \theta$ *in canonical form and any* $\Phi \in (\!| \Delta|\Gamma |\!)$*, the IMJ-automaton* $(\!| \mathcal{C} |\!)_\Phi$ *is canonical.*

Apart from specifying the symbolic stores at each state, the ranking function also determines the shape of configurations that can reach a specific state.

Lemma 42. *If* $\mathcal{A}$ *is canonical then, for all compatible* ρ_0 *and reachable configurations* (q, ρ, σ, H) *in* $(\mathcal{A}, \rho_0)$*, we have* $\mathsf{dom}(\mathsf{rank}(q)) \subseteq \mathsf{dom}(\rho)$*. Moreover, if* q *is not intermediate then* $\mathsf{dom}(\rho) = \mathsf{dom}(\mathsf{rank}(q))$*.*

Proof. Suppose $\hat{q} = (q, \rho, \sigma, H)$ is reachable in $(\mathcal{A}, \rho_0)$ via some run t. We do induction on the length of t. If $\hat{q}$ is initial then the claim holds by definition. Otherwise, let $\hat{q}' = (q', \rho', \sigma', H')$ be the preceding configuration in t. If q is intermediate then the last transition in t is due to some $q' \xrightarrow{\nu X.(\mu^S,\phi)} q$ so $\mathsf{rank}(q) = S$ and, by IH, $\mathsf{dom}(\rho') = \mathsf{dom}(\mathsf{rank}(q'))$. Now, $\mathsf{dom}(S) \subseteq \mathsf{dom}(\rho') \cup X = \mathsf{dom}(\rho)$ so $\mathsf{dom}(\mathsf{rank}(q)) \subseteq \mathsf{dom}(\rho)$.

Suppose q is not intermediate. If q' is not intermediate then by the IH we have that $\mathsf{dom}(\rho') = \mathsf{dom}(\mathsf{rank}(q'))$. If the transition from $\hat{q}'$ to $\hat{q}$ is an ϵ-transition then we can easily see that $\mathsf{dom}(\mathsf{rank}(q)) = \mathsf{dom}(\rho)$. Otherwise, the transition must be due to some

$q' \xrightarrow{\nu X.(\mu^S,\phi)} q$ and $q' \in Q_P$, so $\mathsf{rank}(q) = S$. Moreover, since the transition can be taken, we have $\mathsf{dom}(S) \subseteq \mathsf{dom}(\rho)$. Conversely, we have that $\mathsf{dom}(\rho) = \mathsf{dom}(\rho') \cup X = \mathsf{dom}(\mathsf{rank}(q')) \cup X$ which is included in $\mathsf{dom}(S)$ by the last ranking condition. Finally, suppose that the two preceding configurations in t are $\hat{q}' = (q', \rho', \sigma', H')$ and $\hat{q}'' = (q'', \rho'', \sigma'', H'')$, themselves due to some $q'' \xrightarrow{\nu X'.(\mu^S,\phi)} q' \xrightarrow{X} q$ with $q'' \in Q_O$ and q' intermediate. We have $\mathsf{rank}(q) = S \restriction X$ so $\mathsf{dom}(\mathsf{rank}(q)) = \mathsf{dom}(S) \cap X$. On the other hand, $\mathsf{dom}(\rho) = \mathsf{dom}(\rho') \cap X$, which includes $\mathsf{dom}(S) \cap X$ since the first transition can be taken. Moreover, by first ranking condition, $\mathsf{dom}(S) \subseteq X$, thus $\mathsf{dom}(\mathsf{rank}(q)) = X \supseteq \mathsf{dom}(\rho)$. □

F The decidability procedure

We next use the automata representations of the previous section in order to decide equivalences of IMJ*terms in canonical form. Our starting point are IMJ-automata which represent the game semantics of canonical IMJ* terms, and we proceed as follows.

At a first stage, given deterministic IMJ-automata $\mathcal{A}_1$ and $\mathcal{A}_2$, we construct a *comparison IMJ-automaton* (or *IMJ2-automaton*) $\mathcal{A}'$ such that $\mathcal{A}_1$ and $\mathcal{A}_2$ accept the same play extensions on initial ρ_0 iff the language of $\mathcal{A}'$ (on appropriate ρ_0') is empty.

F.1 IMJ2-automata

For this section, we fix IMJ-automata $\mathcal{A}_1, \mathcal{A}_2$ which are assumed to operate on the same set of symbolic values, moves, stores and labels:

$$\begin{aligned} Val_{\mathtt{S}} &= \mathbb{C} \cup \mathbb{C}_{\mathtt{r}} \\ Mov_{\mathtt{S}} &= Val_{\mathtt{S}}^* \cup \{\mathsf{call}\,\mathtt{r}_i.\mathsf{m}.(\vec{x}), \mathsf{ret}\,\mathtt{r}_i.\mathsf{m}(x) \mid \vec{x} \in Val_{\mathtt{S}}^* \wedge x \in Val_{\mathtt{S}}\} \\ Sto_{\mathtt{S}} &= \mathbb{C}_{\mathtt{r}} \rightharpoonup Intfs \times (Flds \rightharpoonup Val_{\mathtt{S}}) \\ Lab_{\mathtt{S}} &= Mov_{\mathtt{S}} \times Sto_{\mathtt{S}} \end{aligned}$$

and in particular they operate on the same set of registers $\mathbb{C}_{\mathtt{r}} = \{\ \mathtt{r}_i \mid 1 \leq i \leq \mathrm{R}\ \}$.[8] Recall also the sets of transition labels:

$$\begin{aligned} TL_{\mathsf{push}} &= \mathcal{P}(\mathbb{C}_{\mathtt{r}}) \times Lab_{\mathsf{push}} \times \mathbb{C}_{\mathtt{st}} \times \mathcal{P}(\mathbb{C}_{\mathtt{r}}) \\ TL_{\mathsf{pop}} &= \mathcal{P}(\mathbb{C}_{\mathtt{r}}) \times Lab_{\mathsf{pop}} \times \mathbb{C}_{\mathtt{st}} \times \mathcal{P}(\mathbb{C}_{\mathtt{r}})^2 \\ TL_{\mathsf{noop}} &= \mathcal{P}(\mathbb{C}_{\mathtt{r}}) \times Lab_{\mathsf{noop}} \times \{()\} \end{aligned}$$

and $TL = TL_{\mathsf{push}} \uplus TL_{\mathsf{pop}} \uplus TL_{\mathsf{noop}}$, where $Lab_{\mathtt{S}}$ is partitioned as $Lab_{\mathtt{S}} = Lab_{\mathsf{push}} \uplus Lab_{\mathsf{pop}} \uplus Lab_{\mathsf{noop}}$.

We define an extended set of register constants $\mathbb{C}_{\mathtt{r}2}$ and a set $Reg2$ of (extended) register assignments:

$$\begin{aligned} \mathbb{C}_{\mathtt{r}2} &= \{\mathtt{r}_i \mid 1 \leq i \leq 2\mathrm{R}\} \\ Reg2 &= \{\rho : \mathbb{C}_{\mathtt{r}2} \rightharpoonup \mathbb{A} \mid \pi_1(\rho), \pi_2(\rho) \in Reg\} \end{aligned}$$

[8] This restriction is mostly harmless as we can trivially extend the number of registers in an automaton, making sure we never use the additional registers.

where the two projections of ρ are $\pi_1(\rho) = \rho \restriction \mathbb{C}_{\mathtt{r}}$ and

$$\pi_2(\rho) = \{(\mathtt{r}_i, \rho(i + \mathrm{R})) \mid (i + \mathrm{R}) \in \mathsf{dom}(\rho)\};$$

moreover, for each $\rho_1, \rho_2 \in \mathit{Reg}$ let

$$\langle \rho_1, \rho_2 \rangle = \rho_1 \cup \{(\mathtt{r}_{i+\mathrm{R}}, \rho_2(i)) \mid i \in \mathsf{dom}(\rho_2)\}$$

be their pairing in $\mathit{Reg}2$. We also fix a fresh label symbol $\checkmark$ and, for any set X, we write $X^{\checkmark}$ for

$$X^{\checkmark} \;=\; (X \uplus \{\checkmark\})^2 \setminus \{(\checkmark, \checkmark)\}.$$

Stacks are extended via setting

$$\mathit{Stk}2 = (\mathbb{C}_{\mathtt{st}}{}^{\checkmark} \times \mathit{Reg}2)^*$$

and, for each $(s_1, s_2, \rho) :: \sigma \in \mathit{Stk}2$ and $i = 1, 2$, we define,

$$\pi_i((s_1, s_2, \rho) :: \sigma) = (s_i, \pi_i(\rho)) :: \pi_i(\sigma)$$

while $\pi_i(\epsilon) = \epsilon$. Moreover, we pair stacks of equal height by $\langle \epsilon, \epsilon \rangle = \epsilon$ and:

$$\langle (s_1, \rho_1) :: \sigma_1, (s_2, \rho_2) :: \sigma_2 \rangle = (s_1, s_2, \langle \rho_1, \rho_2 \rangle) :: \langle \sigma_1, \sigma_2 \rangle$$

We define automata that combine computations from two constituent IMJ-automata as follows. An ***IMJ2-automaton*** is a tuple $\mathcal{A} = \langle Q, \mathsf{rank}, q_0, X_0, \delta, F \rangle$, where:

- Q is a set of states, partitioned into O- and P-states, and into *normal*, *1-divergent* and *2-divergent* ones:

  $$Q = Q_O \uplus Q_P = Q_N \uplus Q_{D1} \uplus Q_{D2}$$

  and there are given maps $\mathsf{div}_1 : Q_{PN} \to Q_{PD1}$ and $\mathsf{div}_2 : Q_{PN} \to Q_{PD2}$ (we write Q_{PN} for $Q_P \cap Q_N$, etc.);
- $q_0 \in Q_{PN}$ is the initial state, and $F \subseteq Q_{OD}$ is the set of final states;
- $X_0 \subseteq \mathbb{C}_{\mathtt{r}2}$ is the set of initially non-empty registers;
- $\mathsf{rank} : Q \to \{S \in \mathit{Sto}_{\mathtt{S}} \mid \nu_{\mathtt{r}}(S) \subseteq \mathsf{dom}(S)\}^2$ is a ranking function, and we write rank_i for the composition of rank with the i-th projection function ($\pi_i \circ \mathsf{rank}$, $i = 1, 2$);
- $\delta \subseteq (Q_P \times \mathit{TL}_{PO}{}^{\checkmark} \times Q_O) \cup (Q_O \times \mathit{TL}_{OP}{}^{\checkmark} \times Q_P) \cup (Q_O \times (\mathcal{P}(\mathbb{C}_{\mathtt{r}})^{\checkmark} \cup (\mathbb{C}_{\mathtt{r}} \xrightarrow{\cong} \mathbb{C}_{\mathtt{r}})^{\checkmark}) \times Q_O) \cup (Q_P \times \mathcal{P}(\mathbb{C}_{\mathtt{r}})^{\checkmark} \times Q_P)$ is the transition relation, satisfying also:

$$\begin{aligned}\delta \;\subseteq\; & (Q_N \times (\mathit{TL}^2 \cup \mathcal{P}(\mathbb{C}_{\mathtt{r}})^2 \cup (\mathbb{C}_{\mathtt{r}} \xrightarrow{\cong} \mathbb{C}_{\mathtt{r}})^2) \times Q_N) \\ & \cup (Q_{D1} \times (\mathit{TL} \cup \mathcal{P}(\mathbb{C}_{\mathtt{r}}) \cup (\mathbb{C}_{\mathtt{r}} \xrightarrow{\cong} \mathbb{C}_{\mathtt{r}})) \times \{\checkmark\} \times Q_{D1}) \\ & \cup (Q_{D2} \times \{\checkmark\} \times (\mathit{TL} \cup \mathcal{P}(\mathbb{C}_{\mathtt{r}}) \cup (\mathbb{C}_{\mathtt{r}} \xrightarrow{\cong} \mathbb{C}_{\mathtt{r}})) \times Q_{D2})\end{aligned}$$

where $\mathit{TL}_{PO} = \mathit{TL}_{\mathsf{noop}} \cup \mathit{TL}_{\mathsf{push}}$ and $\mathit{TL}_{OP} = \mathit{TL}_{\mathsf{noop}} \cup \mathit{TL}_{\mathsf{pop}}$. In addition, rank satisfies the following conditions.

- $\mathsf{dom}(\mathsf{rank}_1(q_0)) = X_0 \cap \mathbb{C}_{\mathtt{r}}$ and $\mathsf{dom}(\mathsf{rank}_2(q_0)) = \{\mathtt{r}_{i-\mathrm{R}} \mid i \in [\mathrm{R}{+}1, 2\mathrm{R}] \wedge \mathtt{r}_i \in X_0\}$;
- $\mathsf{rank}(Q_{D1}) \subseteq Sto_{\mathtt{S}} \times \{\emptyset\}$ and $\mathsf{rank}(Q_{D2}) \subseteq \{\emptyset\} \times Sto_{\mathtt{S}}$;
- if $q \xrightarrow{X_1,X_2} q'$ in δ then $X_i \subseteq \mathsf{dom}(\mathsf{rank}_i(q))$ and $\mathsf{rank}(q') = (\mathsf{rank}_1(q) \restriction X_1, \mathsf{rank}_2(q) \restriction X_2)$;
- if $q \xrightarrow{X_1,\checkmark} q'$ in δ then $X_1 \subseteq \mathsf{dom}(\mathsf{rank}_1(q))$ and $\mathsf{rank}(q') = (\mathsf{rank}_1(q) \restriction X_1, \emptyset)$; dually for $q \xrightarrow{\checkmark,X_2} q'$ in δ;
- if $q \xrightarrow{\hat{\pi}_1,\hat{\pi}_2} q'$ in δ then $\mathsf{rank}(q') = (\hat{\pi}_1^\dagger \circ \mathsf{rank}_1(q) \circ \hat{\pi}_1^{-1}, \hat{\pi}_2^\dagger \circ \mathsf{rank}_2(q) \circ \hat{\pi}_2^{-1})$;
- if $q \xrightarrow{\hat{\pi}_1,\checkmark} q'$ in δ then $\mathsf{rank}(q') = (\hat{\pi}_1^\dagger \circ \mathsf{rank}_1(q) \circ \hat{\pi}_1^{-1}, \emptyset)$; dually for $q \xrightarrow{\checkmark,\hat{\pi}_2} q'$ in δ;
- if $q \xrightarrow{\nu X_1.(\mu_1^{S_1},\phi_1),\nu X_2.(\mu_2^{S_2},\phi_2)} q'$ in δ then we have $\mathsf{dom}(\mathsf{rank}_i(q)) \cap X_i = \emptyset$, $\mathsf{rank}(q') = (S_1, S_2)$ and if $q \in Q_P$ then $\mathsf{dom}(\mathsf{rank}_i(q)) \subseteq \mathsf{dom}(S_i)$;
- if $q \xrightarrow{\nu X_1.(\mu_1^{S_1},\phi_1),\checkmark} q'$ in δ then $\mathsf{dom}(\mathsf{rank}_1(q)) \cap X_1 = \emptyset$, $\mathsf{rank}(q') = (S_1, \emptyset)$ and if $q \in Q_P$ then $\mathsf{dom}(\mathsf{rank}_1(q)) \subseteq \mathsf{dom}(S_1)$; dually for $q \xrightarrow{\checkmark,\nu X_2.(\mu_2^{S_2},\phi_2)} q'$ in δ.

An IMJ2-automaton as above operates on words over the alphabet $\mathbb{W}$, with configurations given by tuples (q, ρ, σ, H), where $q \in Q$, $\rho \in Reg2$, $\sigma \in Stk2$ and $H \in \mathcal{P}_{\mathsf{fn}}(\mathbb{A})$ such that $\pi_1(\rho)(\mathsf{rank}_1(q)) \cup \pi_2(\rho)(\mathsf{rank}_2(q))$ is a well-defined store. Given an initial $\rho_0 \in Reg2$ such that $\mathsf{dom}(\rho_0) = X_0$, the automaton induces the following configuration graph. The initial configuration is $(q_0, \rho_0, \epsilon, \mathsf{cod}(\rho_0))$. Given configuration (q, ρ, σ, H), let $\Sigma = \pi_1(\rho)(\mathsf{rank}_1(q)) \cup \pi_2(\rho)(\mathsf{rank}_2(q))$ and:

- If $q \xrightarrow{\nu X_1.(\mu_1^{S_1},\phi_1),\nu X_2.(\mu_2^{S_2},\phi_2)} q'$ in δ and, for $i = 1, 2$, $q \xrightarrow{\nu X_i.(\mu_i^{S_i},\phi_i)} q'$ induces $(q, \pi_i(\rho), \pi_i(\sigma), H) \xrightarrow{m_i^{\Sigma_i}} (q', \rho_i, \sigma_i, H_i)$ on an (ordinary) IMJ-automaton and:
  1. $m_1 = m_2$,[9]
  2. if $q \in Q_P$ then $\Sigma[\Sigma_1] \cup \Sigma[\Sigma_2]$ is well-defined,[10]
  3. if $q \in Q_O$ then $\Sigma_1 \cup \Sigma_2$ is well-defined,

  then $(q, \rho, \sigma, H) \xrightarrow{m_1^{\Sigma_1 \cup \Sigma_2}} (q', \langle \rho_1, \rho_2 \rangle, \langle \sigma_1, \sigma_2 \rangle, H_1 \cup H_2)$.
- If $q \xrightarrow{\nu X_1.(\mu_1^{S_1},\phi_1),\nu X_2.(\mu_2^{S_2},\phi_2)} q'$ in δ, with $q \in Q_{PN}$, and $q \xrightarrow{\nu X_1.(\mu_1^{S_1},\phi_1)} q'$ induces a transition $(q, \pi_1(\rho), \pi_1(\sigma), H) \xrightarrow{m_1^{\Sigma_1}} \hat{q}_1$ on an IMJ-automaton and $q \xrightarrow{\nu X_2.(\mu_2^{S_2},\phi_2)} q'$ induces no transition $(q, \pi_1(\rho), \pi_1(\sigma), H) \xrightarrow{m_2^{\Sigma_2}} \hat{q}_2$ such that conditions 1,2 above be satisfied, then the automaton *1-diverges*, that is, $(q, \rho, \sigma, H) \xrightarrow{\epsilon} (\mathsf{div}_1(q), \langle \pi_1(\rho), \emptyset \rangle, \sigma, H)$.
- Dually to the case above, if the first component cannot replicate the move of the second one, then we perform a *2-divergence* transition.

[9] because IMJ-automata are of the visible pushdown kind, $m_1 = m_2$ implies that ϕ_1, ϕ_2 are instructions of the same kind (push, pop, noop).

[10] i.e. we stipulate that Σ_1, Σ_2 agree on common names, while values of private names remain unchanged. This stems from the fact that we do not compare plays but, rather, representations thereof.

– If $q \xrightarrow{\nu X_1.(\mu_1^{S_1},\phi_1),\checkmark} q'$ in δ (so $q, q' \in Q_{D1}$) inducing $(q, \pi_1(\rho), \pi_1(\sigma), H) \xrightarrow{m_1^{\Sigma_1}}$ $(q', \rho_1, \sigma_1, H_1)$ on an IMJ-automaton, then $(q, \rho, \sigma, H) \xrightarrow{m_1^{\Sigma_1}} (q', \langle \rho_1, \emptyset \rangle, \langle \sigma_1, \sigma_2 \rangle, H_1)$ where (setting $z = (\mu_1^{S_1}, \phi_1)$):

$$\sigma_2 = \begin{cases} (\checkmark, \emptyset) :: \pi_2(\sigma) & z \in Lab_{\mathsf{push}} \\ \sigma' & z \in Lab_{\mathsf{pop}} \wedge \pi_2(\sigma) = x :: \sigma' \\ \pi_2(\sigma) & z \in Lab_{\mathsf{noop}} \end{cases}$$

– For each $q \xrightarrow{\checkmark,\nu X_2.(\mu_2^{S_2},\phi_2)} q'$ in δ we do the analogous of the dual case above.
– If $q \xrightarrow{X_1,X_2} q'$ in δ then $(q, \rho, \sigma, H) \xrightarrow{\epsilon} (q', \langle \rho_1, \rho_2 \rangle, \sigma, H)$ with $\rho_i = \pi_i(\rho) \restriction X_i$.
– If $q \xrightarrow{\pi_1',\pi_2'} q'$ in δ then $(q, \rho, \sigma, H) \xrightarrow{\epsilon} (q', \langle \rho_1, \rho_2 \rangle, \sigma, H)$ with $\rho_i = \pi_i(\rho) \circ \pi_i'^{-1}$.
– If $q \xrightarrow{X_1,\checkmark} q'$ in δ then $(q, \rho, \sigma, H) \xrightarrow{\epsilon} (q', \langle \rho_1, \pi_2(\rho) \rangle, \sigma, H)$ with $\rho_1 = \pi_1(\rho) \restriction X_1$. Dually if $q \xrightarrow{\checkmark,X_2} q'$ in δ.
– If $q \xrightarrow{\pi_1',\checkmark} q'$ in δ then $(q, \rho, \sigma, H) \xrightarrow{\epsilon} (q', \langle \rho_1, \pi_2(\rho) \rangle, \sigma, H)$ with $\rho_1 = \pi_1(\rho) \circ \pi_1'^{-1}$. Dually if $q \xrightarrow{\checkmark,\pi_2'} q'$ in δ.

Note in particular that if $\mathcal{A}$ only contains transitions which include $\checkmark$ then it operates an IMJ-automaton with a ranking function. In fact, any canonical IMJ-automaton can be rendered into an IMJ2-automaton by simply changing each transition (q, z, q') to $(q, z, \checkmark, q')$ and adding an initial state with a sole transition that causes 1-divergence.

Suppose now $\mathcal{A}_i = \langle Q_i, q_{0i}, X_{0i}, \delta_i, F_i \rangle$, for $i = 1, 2$, are our initial canonical IMJ-automata with ranking functions $\mathsf{rank}^1, \mathsf{rank}^2$ respectively. Moreover, assume that $\mathcal{A}_i$ have been instrumented so that every P-state has exactly one outgoing transition (e.g. by adding dummy transitions to sink states). We construct the IMJ2-automaton $\mathcal{A}_1 \otimes \mathcal{A}_2 = \langle Q, \mathsf{rank}, q_0, X_0, \delta, F \rangle$ by taking:

– $Q = (Q_{1O} \times Q_{2O}) \uplus (Q_{1P} \times Q_{2P}) \uplus Q_1 \uplus Q_2$ with

$$\begin{aligned} Q_O &= (Q_{1O} \times Q_{2O}) \cup Q_{1O} \cup Q_{2O} \\ Q_P &= (Q_{1P} \times Q_{2P}) \cup Q_{1P} \cup Q_{2P} \\ Q_N &= (Q_{1O} \times Q_{2O}) \cup (Q_{1P} \times Q_{2P}) \\ Q_D &= Q_1 \cup Q_2 \end{aligned}$$

  and $\mathsf{div}_1, \mathsf{div}_2$ being the two projection functions;
– $q_0 = (q_{01}, q_{02})$, $X_0 = X_{01} \cup \{\mathtt{r}_{i+\mathrm{R}} \mid \mathtt{r}_i \in X_{02}\}$ and $F = F_1 \cup F_2$;
– $\mathsf{rank} = \{((q_1, q_2), (\mathsf{rank}^1(q_1), \mathsf{rank}^2(q_2))) \mid (q_1, q_2) \in Q\}$
  $\cup \{(q_1, (\mathsf{rank}^1(q_1), \emptyset)) \mid q_1 \in Q_1\}$
  $\cup \{(q_2, (\emptyset, \mathsf{rank}^1(q_2))) \mid q_2 \in Q_2\}$;
– for each $q_1 \in Q_1$ and $q_2 \in Q_2$ we include in δ' precisely the following transitions:
  - if $q_i \xrightarrow{\nu X_i.(\mu_i^{S_i},\phi_i)} q_i'$ in δ_i, for $i = 1, 2$, then add $(q_1, q_2) \xrightarrow{\nu X_1.(\mu_1^{S_1},\phi_1),\nu X_2.(\mu_2^{S_2},\phi_2)} (q_1', q_2')$ in δ';

- if $q_1 \xrightarrow{\nu X_1.(\mu_1^{S_1},\phi_1)} q_1'$ in δ_1 then $q_1 \xrightarrow{\nu X_1.(\mu_1^{S_1},\phi_1),\checkmark} q_1'$;
- if $q_2 \xrightarrow{\nu X_2.(\mu_2^{S_2},\phi_2)} q_2'$ in δ_2 then $q_2 \xrightarrow{\checkmark,\nu X_2.(\mu_2^{S_2},\phi_2)} q_2'$;
- if $q_1 \xrightarrow{X_1} q_1'$ in δ_1 then $(q_1,q_2) \xrightarrow{X_1,\mathsf{dom}(\mathsf{rank}(q_2))} (q_1',q_2)$ and $q_1 \xrightarrow{X_1,\checkmark} q_1'$;
- if $q_1 \xrightarrow{\pi_1} q_1'$ in δ_1 then $(q_1,q_2) \xrightarrow{\pi_1,\mathsf{id}} (q_1',q_2)$ and $q_1 \xrightarrow{\pi_1,\checkmark} q_1'$;
- if $q_2 \xrightarrow{X_2} q_2'$ in δ_2 then $(q_1,q_2) \xrightarrow{\mathsf{dom}(\mathsf{rank}(q_1)),X_2} (q_1,q_2')$ and $q_2 \xrightarrow{\checkmark,X_2} q_2'$;
- if $q_2 \xrightarrow{\pi_2} q_2'$ in δ_2 then $(q_1,q_2) \xrightarrow{\mathsf{id},\pi_2} (q_1,q_2')$ and $q_2 \xrightarrow{\checkmark,\pi_2} q_2'$.

We can see that, by construction, $\mathcal{A}_1 \otimes \mathcal{A}_2$ simulates the two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ as long as they agree on the accepted input (modulo extension). As soon as they disagree, $\mathcal{A}_1 \otimes \mathcal{A}_2$ will switch to divergence mode and try to complete the input that has been accepted so far to a complete play (up to extension). Such a play would be a witness of inequality of the languages accepted by the two automata.

Lemma 43. *Let $\Delta|\Gamma \vdash \mathcal{C}_1, \mathcal{C}_2 : \theta$ be IMJ^* terms in canonical form, suppose (m^Σ, Φ, ρ_0) is compatible and let $\mathcal{A}_i = (\!|\mathcal{C}_i|\!)_\Phi$ $(i = 1,2)$, $\mathcal{A}' = \mathcal{A}_1 \otimes \mathcal{A}_2$ and $\rho_0' = \langle \rho_0, \rho_0 \rangle$. Then $L(\mathcal{A}', \rho_0')$ is empty iff: $\mathsf{comp}([\![\Delta|\Gamma \vdash \mathcal{C}_1 : \theta]\!]) \cap P^{m,\Sigma}_{[\![\Delta|\Gamma\vdash\theta]\!]} = \mathsf{comp}([\![\Delta|\Gamma \vdash \mathcal{C}_2 : \theta]\!]) \cap P^{m,\Sigma}_{[\![\Delta|\Gamma\vdash\theta]\!]}$.*

Proof. Suppose $w \in L(\mathcal{A}', \rho_0')$ and assume WLOG that w is accepted via 1-divergence. Then, by construction of $\mathcal{A}'$, the accepting run for w yields an accepting run for w in $\mathcal{A}_1$ via first projection. Hence, $w \in L(\mathcal{A}_1, \rho_0)$ and, therefore, $\mathsf{ext}(m^\Sigma w) \in \mathsf{comp}([\![\Delta|\Gamma \vdash \mathcal{C}_1 : \theta]\!])$ by Theorem 17. On the other hand, via second projection of the accepting run for w up to the point of switching to divergence mode, we obtain a run for $\mathcal{A}_2$ which, however, may only be resumed in $\mathcal{A}_2$ by a different (up to extension) P-move than that required by w. Hence, $\mathsf{ext}(m^\Sigma w) \notin [\![\Delta|\Gamma \vdash \mathcal{C}_2 : \theta]\!]$. Conversely, let $m^\Sigma w \in \mathsf{comp}([\![\Delta|\Gamma \vdash \mathcal{C}_1 : \theta]\!]) \setminus [\![\Delta|\Gamma \vdash \mathcal{C}_2 : \theta]\!]$ and let $m^\Sigma w' x y$ be its least prefix that appears in $[\![\Delta|\Gamma \vdash \mathcal{C}_1 : \theta]\!] \setminus [\![\Delta|\Gamma \vdash \mathcal{C}_2 : \theta]\!]$. By construction, our automata are closed under legal O- to P-transitions, so $\mathcal{A}_2$ must accept (a representation of) $m^\Sigma w' x$ and fail to process y. On the other hand, $\mathcal{A}_1$ can process a whole representation of $m^\Sigma w$. Thus, $\mathcal{A}'$ will operate as product automaton until $m^\Sigma w' x$ and then enter 1-divergence mode and continue as $\mathcal{A}_1$. Hence, $m^\Sigma \tilde{w} \in L(\mathcal{A}', \rho_0')$ for some $\mathsf{ext}(m^\Sigma \tilde{w}) \ni m^\Sigma w$. □

We observe that the constructed $\mathcal{A}'$ is deterministic, since $\mathcal{A}_1$ and $\mathcal{A}_2$ are and transitions to divergence mode are input-directed. Moreover, we can simulate the operation of an IMJ2-automaton by an IMJ-automaton $\mathcal{B}$ which keeps in its registers all the names appearing in $\mathcal{A}'$ (once each) and stores in its state: the current state of $\mathcal{A}'$; the rank of the state according to the ranking function of $\mathcal{A}'$; information on how the names of $\mathcal{A}'$ are distributed in the registers of $\mathcal{B}$ (i.e. a partial map $\chi : [1, 2\mathrm{R}] \rightharpoonup [1, \mathrm{R}]$ such that $\chi \restriction [1, \mathrm{R}]$ and $\chi \restriction [\mathrm{R}{+}1, 2\mathrm{R}]$ are injective). The latter two components are enough for statically determining the IMJ2 preconditions for taking a given transition. These preconditions are the essential difference between IMJ2 and IMJ automata: removing name repetitions from registers is otherwise standard (e.g. [15, Section 6]). This yields Theorem 19. Decidability then follows from the fact that IMJ-automata (and, more generally, pushdown register automata with globally fresh transitions) are decidable for emptiness, by reduction to emptiness for pushdown register automata [10].

G Termination for first-order recursion

In the rest of this section we will show that the class of 1-recursive terms $\Delta|\emptyset \vdash P : \mathsf{void}$, where Δ allows for G-valued fields only, have depth-bounded visible state. We will use the fact that while is encodable in this fragment to simplify the argument, by assuming that no 1-recursive term includes any occurrence of while.

We shall analyse the visible state of computations in this fragment by considering the length of the chains of object names by which they are supported.

Definition 44. Given a state S, a ***support chain*** in S is a sequence of distinct object names $a_1 \cdots a_n$ drawn from $\mathsf{dom}(S)$, for which $a_{i+1} \in \nu(S(a_i))$ for all $1 \leq i < n$. Due to the recursive types restriction, we are mainly interested in ***method support chains*** which are support chains in which each pair $a_i a_{i+1}$ is justified by a_{i+1} occurring in a method of a_i, i.e. there is some method identifier m such that $a_{i+1} \in \nu(S(a_i).m)$. Define $C(S)$ as the maximum length of any method support chain in S.

The length of the largest support chain for a state S gives exactly the closure ordinal required for the inductive definition of visible state. Due to the recusive types restriction, chains arising as a result of objects stored in fields are inheritently bounded by the size of the interface table. Hence, we shall exclusively be interested in bounding the length of method support chains. We start by showing that, if a term makes only first-order calls then the length of such chains can grow by at most one from initial to final state. In the following let us write A to denote any context whose domain consists entirely of object names.

Lemma 45. *Fix a term $\Delta|A \vdash P : \mathsf{void}$ where Δ allows for G-valued fields only. If $(S,\, P,\, \epsilon) \longrightarrow^* (S',\, P',\, F')$ making only first-order method calls then:*

(i) $C(S') \leq C(S) + 1$
(i) For every subterm N of P' or F' which has shape $\mathsf{new}\,\{z : I;\, \mathcal{M}\}$, $C(S'@N) \leq C(S)$.

Proof. The proof is by induction on the length of transition sequence. If the sequence has length 0 then the result is obvious. In the step case, assume that the conclusion holds for $(S,\, P,\, \epsilon) \longrightarrow^* (S'',\, P'',\, F'')$ and consider $(S'',\, P'',\, F'') \longrightarrow (S',\, P',\, F')$. The only transitions that intoduce names of method carrying objects into S', P' and F' are the rules for the evaluation of object creation expressions, method calls and method returns (recall that fields cannot store method carrying objects):

In case the transition is a method call, P'' is of shape $E[a.m(\vec{v})]$ for some implementation $\lambda\vec{x}.\, M$, $P' = M[\vec{v}/\vec{x}]$ and $F' = E :: F''$. Clause (i) follows immediately because $S' = S''$. By assumption (i) a has method support chain of length bounded by $C(S) + 1$, so $C(S''@M)$ is bounded by $C(S)$. By the assumption on the lemma, the method call is first-order so that any objects in $\vec{v}$ do not carry methods. Hence, also any object creation expression N inside P' has $C(S'@N) \leq C(S)$. Furthermore, by assumption (ii), any object creation expression N in E or F'' has $C(S'@N) \leq C(S)$ so the same is true of the same expressions in $E :: F''$.

In case the transition is an object creation expression, P'' is of shape $E[\mathsf{new}\,\{z : I;\, \mathcal{M}\}]$, $P' = E[a]$, $S' = S'' \uplus \{(a,\, I,\, (V_I,\, \mathcal{M}[a/z]))\}$ and $F'' = F'$. By assumption

(ii), $C(S''@\mathcal{M}) \leq C(S)$, hence $C(S'@a) \leq C(S) + 1$ as required by (i). To see that clause (ii) holds, let N be an object creation expression in $E[a]$ or F'. In the latter case, N is also in F'' and the result holds by assumption (ii). In the former case N is necessarily in a part of E disjoint from a, since evaluation contexts do not have holes within object creation expressions. Hence, N is also in P'' and the result holds by assumption (ii).

In case the transition is a method return, P'' is of shape v, F'' is of shape $E :: F'$, and $P' = E[v]$. Clause (i) follows immediately because $S' = S''$. To see that clause (ii) holds assume that N is an object creation expression in $E[v]$ or F'. In the latter case, the result holds by assumption (ii). In the former case, since evaluation contexts do not have holes inside object creation expressions, N was also present in E and the result holds by assumption (ii). □

When a term does make a method call that is not first-order, it is able to pass a method-carrying object a into a new stack frame. Due to the mechanism for method calls, a can be substituted for some formal parameter x which may occur inside the body of an object creation expression. Hence, when this expression is evaluated the newly created object will have a method support chain which is that of a extended by 1. This process can be repeated by growing an ever-longer call chain, but it is ultimately bounded by the 1-recursion restriction imposed on terms in this fragment. Hence, we will build a well-founded measure out of this restriction.

Definition 46. Fix a 1-recursive term $\Delta|\emptyset \vdash P : \mathsf{void}$. For each method $I.m$ declared in Δ, define a measure $\#I.m$ using the method dependency graph of the term: $\#I.m$ is 0 if it is not possible to reach a higher-order method $(J,\, m')$ from $(I,\, m)$ and, otherwise, $\#I.m$ is the greatest length of any simple path from $(I,\, m)$ to a higher-order method.

Note that, since the graph may contain ground cycles, if $(I,\, m)$ has an edge to $(J,\, m')$ then there are three cases:

(i) If $(I,\, m)$ cannot reach a higher-order method then neither can $(J,\, m')$ and hence $\#J.m' = 0 = \#I.m$.
(ii) If $(I,\, m)$ can reach a higher-order method, and there is a path which does not contain $(I,\, m)$ from $(J,\, m')$ to a higher-order method, then $\#J.m' < \#I.m$.
(iii) If $(I,\, m)$ can reach a higher-order method, but every path from $(J,\, m')$ to a higher-order method contains $(I,\, m)$, then $\#J.m' > \#I.m$. In this case, $(I,\, m)$ and $(J,\, m')$ occur on a cycle and hence are first-order methods.

Given two terms $\Delta|\Gamma \vdash N : \theta$ and $\Delta|A \vdash N' : \theta$ we say that N' is an *instance* of N just if N' can be obtained from N by substituting free variables by values of the same type.

Lemma 47. *Fix a 1-recursive term $\Delta|A \vdash P : \mathsf{void}$, where Δ allows for G-valued fields only. Let $\lambda x.\, N$ be an implementation in P of some method $I.m$ and let N' be an instance of N. Then, if $(S,\, N',\, \epsilon) \longrightarrow^* (S',\, N'',\, F)$, it follows that:*

$$C(S'@N'') \leq 2|P|^{\#I.m} + 2|P|^{\#I.m-1} \cdots + 2|P| + 1 + C(S@N')$$

Proof. The proof is by induction on the measure. When $\#I.m = 0$, there are no calls to higher-order methods. Hence, it follows from Lemma 45 that $C(S'@N'')$ is bounded by $C(S@N')+1$. When $\#I.m > 0$, then the shape of the transition sequence is generally:

$$\begin{array}{rll}
(S_0,\, N',\, \epsilon) & \longrightarrow^* (S_1,\, E_1[a_1.m_1(\vec{v_1})],\, \epsilon) & (1)\\
& \longrightarrow^* (S_1',\, E_1[v_1],\, \epsilon) & (1')\\
& \longrightarrow^* (S_2,\, E_2[a_2.m_2(\vec{v_2})],\, \epsilon) & (2)\\
& \longrightarrow^* (S_2',\, E_2[v_2],\, \epsilon) & (2')\\
& \longrightarrow^* \cdots & \\
& \longrightarrow^* (S_n,\, E_n[a_n.m_n(\vec{v_n})],\, \epsilon) & (n)\\
& \longrightarrow^* (S_n',\, E_n[v_n],\, \epsilon) & (n')\\
& \longrightarrow^* (S',\, N'',\, \epsilon) &
\end{array}$$

with $S_0 = S$ and (i') labelling the subsequence that ends with the matching return from the call at the end of the subsequence labelled by (i). Furthermore, the calls $(1), \ldots, (n)$ the only calls made with empty call stack. This is the case when all calls made in the empty call stack successfully return. If a call does not return then (there are no further calls in the empty call stack) and the shape ends with:

$$\begin{array}{ll}
\longrightarrow^* (S_n,\, E_n[a_n.m_n(\vec{v_n})],\, \epsilon)\ (n)\\
\longrightarrow^* (S',\, N'',\, F)
\end{array}$$

Let us first discuss the case when all calls return. Each subsequence labelled (i) contains no calls and consequently, it follows from Lemma 45 that $C(S_i@E[a_i.m_i(\vec{v_i})]) \leq C(S_{i-1}'@E[v_{i-1}])+1$. Similarly, the final subsequence contains no calls and hence has $C(S'@N'') \leq C(S_n'@E[v_n]) + 1$ Each subsequence labelled (i') is the evaluation of a call to some method $I_i.m_i$ and so starts from an instance of an implementation of this method. Either $\#I_i.m_i \geq \#I.m$ because the method is part of a cycle in the dependency graph or it has measure bounded above by $\#I.m - 1$. In the former case, the method is first-order and so $C(S_i'@E[v_i]) = C(S_i'@E[]) \leq C(S_i@E[a_i.m_i(\vec{v_i})])$. In the latter case it follows from the induction hypothesis that $C(S_i'@E[v_i])$ is bounded above by:

$$2|P|^{\#I.m-1} + 2|P|^{\#I.m-2} \cdot 2|P| + 1 + C(S_i@E[a_i.m_i(\vec{v_i})])$$

Hence, we take this latter, larger value as an upper bound. In summary then, since n calls occur in the empty stack context, we obtain that $C(S'@N'')$ is bounded above by:

$$(n \cdot (2|P|^{\#I.m-1} + \cdots + 2|P| + 1)) + (n+1) + C(S@N')$$

Since (in the absence of the while construct) the number of calls that can occur in a given stack frame is limited by the number of call subterms of shape $M.m(\vec{M'})$, it follows that $n \leq |P|$. Hence

$$C(S'@N'') \leq 2|P|^{\#I.m} + 2|P|^{\#I.m-1} \cdots + 2|P| + 1 + C(S@N')$$

as required. In the case when the final call does not return, there is no final subsequence of transitions in the empty call stack and hence the size of the longest method support chain in S' is at least one smaller. □

Corollary 48. *There is a function $f : \mathbb{N} \to \mathbb{N}$ such that, for any 1-recursive term $\Delta|\emptyset \vdash P : \mathsf{void}$ (where all fields in Δ are* G*-valued) and any transition sequence $(\emptyset, P, \epsilon) \longrightarrow^* (S, M, F)$, it follows that $C(S@M) \leq f(|P|)$.*

Proof. We first reformulate P so that it is the body of some method. Hence, we consider the (visible state equivalent) term $\mathsf{new}\,\{z : I;\ run : \lambda z'.\, P\}.run()$ for some new interface $I = (run : \mathsf{void} \to \mathsf{void})$ and new variables z and z'. Then observe that the largest simple path from any method (I, m) in the method dependency graph is at most $|P|$. □

By viewing the recursive computation of the visible state $S@M$ of some state S relative to a term M as a tree, we see that the length of any path in the tree is bounded by the length of support chains and the branching factor at each object name a is bounded by the size of $\nu(S(a))$. Since we are now able to give a bound on the length of such chains all that remains is to observe that $S(a)$ is always an instance of a subterm of the program.

Corollary 49. *Fix a 1-recursive term $\Delta|\emptyset \vdash P : \mathsf{void}$, where Δ allows for* G*-valued fields only. Then P has depth-bounded visible state.*

Proof. Let $(\emptyset, P, \epsilon) \longrightarrow^* (S, M, F)$ and consider any support chain $a_1 \cdots a_k$ in $S@M$. Since no fields store method-carrying objects, it follows that if any pair $a_i a_{i+1}$ is justified by a field of a_i storing a_{i+1}, then a_{i+1} does not carry methods and hence the length of the suffix $a_i \cdots a_k$ is bounded by a function depending only on $|\Delta|$. So let the prefix $a_1 \cdots a_{i-1}$ consist only of method carrying objects. Then it follows from Corollary 48 that this prefix, a method support chain, is bounded by a function depending only on $|P|$. Hence, the support chain as a whole is bounded by a function depending only on $|P|$ and $|\Delta|$. Since, for each name $a \in \mathsf{dom}(S)$, $\nu(S(a))$ is bounded in size by $S(a)$, and $S(a)$ is necessarily an instance of a subterm of P, it follows that $\nu(S(a))$ is bounded in size by $|P|$. Hence, since the size of any support chain is bounded by a function depending only on $|P|$ and $|\Delta|$, and the branching factor of names in the chain is bounded by $|P|$, so the number of names in $S@M$ is bounded by a function depending only $|P|$ and $|\Delta|$. □

Corollary 50. *Fix a 1-recursive term $\Delta|\emptyset \vdash P : \mathsf{void}$, where all fields in Δ are* G*-valued. Then $P{\Downarrow}$ is decidable.*

Proof. During any evaluation of P, methods attached to objects in the visible state are always instances of subterms of P. Since there are only finitely many subterms and since P has depth-bounded visible state, it follows that there is a bound on the number of shapes of visible state, modulo the specific choice of object names. Hence, we follow [3] in representing the computation as a pushdown system. □