Algorithmic games for full ground references

A. S. Murawski[1] and N. Tzevelekos[2,⋆]

[1] University of Leicester
[2] Queen Mary, University of London

Abstract. We present a full classification of decidable and undecidable cases for contextual equivalence in a finitary ML-like language equipped with full ground storage (both integers and reference names can be stored). The simplest undecidable type is unit $\rightarrow$ unit $\rightarrow$ unit. At the technical level, our results marry game semantics with automata-theoretic techniques developed to handle infinite alphabets. On the automata-theoretic front, we show decidability of the emptiness problem for register pushdown automata extended with fresh-symbol generation.

1 Introduction

Mutable variables in which numerical values can be stored for future access and update are the pillar of imperative programming. The memory in which the values are deposited can be allocated statically, typically to coincide with the lifetime of the defining block, or dynamically, on demand, with the potential to persist forever. In order to support memory management, modern programming languages feature mechanisms such as *pointers* or *references*, which allow programmers to access memory via addresses. Languages like C (through `int*`) or ML (via `int ref ref`) make it possible to store the addresses themselves, which creates the need for storing references to references etc. We refer to this scenario as *full ground storage*. In this paper we study an ML-like language GRef with full ground storage, which permits the creation of references to integers as well as references to integer references, and so on.

We concentrate on contextual equivalence[3] in that setting. Reasoning about program equivalence has been a central topic in programming language semantics since its inception. This is in no small part due to important applications, such as verification problems (equivalence between a given implementation and a model implementation) and compiler optimization (equivalence between the original program and its transform). Specifically, we attack the problem of automated reasoning about our language in a finitary setting, with finite datatypes and with looping instead of recursion, where decidability questions become interesting and the decidability/undecidability frontier can be identified. In particular, it is possible to quantify the impact of higher-order types on decidability, which goes unnoticed in Turing-complete frameworks.

The paper presents a complete classification of cases in which GRef program equivalence is decidable. The result is phrased in terms of the syntactic shape of types. We

⋆ Supported by a Royal Academy of Engineering Research Fellowship.

[3] Two program phrases are regarded as *contextually equivalent*, or simply *equivalent*, if they can be used interchangeably in any context without affecting the observable outcome.

write $\theta_1, \cdots, \theta_k \vdash \theta$ to refer to the problem of deciding contextual equivalence between two terms M_1, M_2 such that $x_1 : \theta_1, \cdots, x_m : \theta_m \vdash M_i : \theta$ $(i = 1, 2)$. We investigate the problem using a fully abstract game model of GRef.[4] Such a model can be easily obtained by modifying existing models of more general languages, e.g. by either adding type information to Laird's model of untyped references [14] or trimming down our own model for general references [18]. The models are *nominal* [1,14] in that moves may involve elements from an infinite set of *names* to account for reference names. Additionally, each move is equipped with a store whose domain consists of all names that have been revealed (played) thus far and the corresponding values. Note that values of reference types also become part of the domain of the store. This representation grows as the play unfolds and new names are encountered. We shall rely on the model both for decidability and undecidability results. Our work identifies the following undecidable cases as minimal.

$$\begin{array}{ll} \vdash \mathsf{unit} \to \mathsf{unit} \to \mathsf{unit} & (\mathsf{unit} \to \mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit} \vdash \mathsf{unit} \\ \vdash ((\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \to \mathsf{unit} & (((\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \to \mathsf{unit}) \to \mathsf{unit} \vdash \mathsf{unit} \end{array}$$

Obviously, undecidability extends to typing judgments featuring syntactic supertypes of those listed above (for instance, when fourth-order types appear on the left-hand side of the turnstile or types of the shape $\theta_1 \to \theta_2 \to \theta_3$ occur on the right). The remaining cases are summarized by typing judgements in which each of $\theta_1, \cdots, \theta_m$ is generated by the grammar given on the left below, and θ by the grammar on the right,

$$\Theta_L ::= \beta \mid \Theta_R \to \Theta_L \qquad\qquad \Theta_R ::= \beta \mid \Theta_1 \to \beta$$

where β stands any ground type and Θ_1 is a first-order type, i.e. $\beta ::= \mathsf{unit} \mid \mathsf{int} \mid \mathsf{ref}^i\, \mathsf{int}$ and $\Theta_1 ::= \beta \mid \beta \to \Theta_1$. We shall show that all these cases are in fact decidable. In order to arrive at a decision procedure we rely on effective reducibility to a canonical (β-normal) form. These forms are then inductively translated into a class of automata over infinite alphabets that represent the associated game semantics. Finally, we show that the representations can be effectively compared for equivalence.

The automata we use are especially designed to read moves-with-stores in a single computational step. They are equipped with a finite set of registers for storing elements from the infinite alphabet (names). Moreover, in a single transition step, the content of a subset of registers can be pushed onto the stack (along with a symbol from the stack alphabet), to be popped back at a later stage. We use visibly pushdown stacks [4], i.e. the alphabet can be partitioned into letters that consistently trigger the same stack actions (push, pop or no-op). Conceptually, the automata extend register pushdown automata [6] with the ability to generate fresh names, as opposed to their existing capability to generate names not currently present in registers. Crucially, we can show that the emptiness problem for the extended machine model remains decidable.

Because the stores used in game-semantic plays can grow unboundedly, one cannot hope to construct the automata in such a way that they will accept the full game semantics of terms. Instead we construct automata that, without loss of generality, will accept plays in which the domains of stores are bounded in size. Each such restricted

[4] A model is *fully abstract* if it captures contextual equivalence denotationally, i.e. equivalence can be confirmed/disproved by reference to the interpretations of terms.

play can be taken to represent a *set* of real plays compatible with the representation. Compatibility means that values of names omitted in environment-moves (O-moves) can be filled in arbitrarily, but values of names omitted in program-moves (P-moves) must be the same as in preceding O-moves. That is to say, the omissions leading to bounded representation correspond to copy-cat behaviour.

Because we work with representations of plays, we cannot simply use off-the-shelf procedures for checking program equivalence, as the same plays can be represented in different ways: copy-cat behaviour can be modelled explicitly or implicitly via the convention. However, taking advantage of the fact that stacks of two visibly pushdown automata over the same partitioning of the alphabet can be synchronized, we show how to devise another automaton that can run automata corresponding to two terms in parallel and detect inconsistencies in the representations of plays. Exploiting decidability of the associated emptiness problem, we can conclude that GRef program equivalence in the above-mentioned cases is decidable.

Related Work. The investigations into models and reasoning principles for storage have a long history. In this quest, storage of names was regarded by researchers as an indispensable intermediate step towards capturing realistic languages with dynamic-allocated storage, such as ML or Java. Relational methods and environmental bisimulations for reasoning about program equivalence in settings similar to ours were studied in [20,5,13,3,7,21], albeit without decidability results. More foundational work included labelled transition system semantics [11] and game semantics [14,18]. In both cases, it turned out that the addition of name storage simplified reasoning, be it bisimulation-based or game-semantic. In the former case, bisimulation was even unsound without full ground storage. In the latter case, the game model of integer storage [16] turned out more intricate (complicated store abstractions) than that for full ground or general storage [14,18]. As for decidability results, finitary Reduced ML (integer storage only) was studied by us in [17], yet only judgements of the form $\cdots, \beta \to \beta, \cdots \vdash \beta$ were tackled due to intricacies related to store abstractions (in absence of full ground storage, names cannot be remembered by programs). A closely related language, called RML [2] (integer storage but with *bad references*, that is, constructs of reference type which do not correspond to valid reference cells) was studied in [15,10], but no full classification has emerged yet. In particular, although the class of types shown decidable is common in both languages, the status of the types that we list as undecidable above remains open in the case of RML.

2 GRef

We work with a finitary ML-like language GRef whose types θ are generated according to the following grammar.

$$\theta ::= \beta \mid \theta \to \theta \qquad \beta \ ::= \ \mathsf{unit} \mid \gamma \qquad \gamma \ ::= \ \mathsf{int} \mid \mathsf{ref}\,\gamma$$

Note that reference types are available for each type of the shape γ (full ground storage). The language is best described as the call-by-value λ-calculus over the ground types β augmented with finitely many constants, do-nothing command, case distinction, looping, and reference manipulation (allocation, dereferencing, assignment). The typing

$$\frac{}{\Gamma \vdash () : \mathsf{unit}} \qquad \frac{i \in \{0, \cdots, max\}}{\Gamma \vdash i : \mathsf{int}} \qquad \frac{(x : \theta) \in \Gamma}{\Gamma \vdash x : \theta} \qquad \frac{\Gamma \vdash M : \mathsf{int} \quad \Gamma \vdash N : \mathsf{unit}}{\Gamma \vdash \mathsf{while}\, M\, \mathsf{do}\, N : \mathsf{unit}}$$

$$\frac{\Gamma \vdash M : \mathsf{int} \quad \Gamma \vdash N_0 : \theta \quad \cdots \quad \Gamma \vdash N_{max} : \theta}{\Gamma \vdash \mathsf{case}(M)[N_0, \cdots, N_{max}] : \theta} \qquad \frac{\Gamma \vdash M : \theta \to \theta' \quad \Gamma \vdash N : \theta}{\Gamma \vdash MN : \theta'}$$

$$\frac{\Gamma \cup \{x : \theta\} \vdash M : \theta'}{\Gamma \vdash \lambda x^{\theta}.M : \theta \to \theta'} \qquad \frac{\Gamma \vdash M : \gamma}{\Gamma \vdash \mathsf{ref}_{\gamma}(M) : \mathsf{ref}\,\gamma} \qquad \frac{\Gamma \vdash M : \mathsf{ref}\,\gamma}{\Gamma \vdash !M : \gamma} \qquad \frac{\Gamma \vdash M : \mathsf{ref}\,\gamma \quad \Gamma \vdash N : \gamma}{\Gamma \vdash M := N : \mathsf{unit}}$$

Fig. 1. Syntax of GRef.

rules are given in Figure 1. In what follows, we write $M; N$ for the term $(\lambda z^{\theta}.N)M$, where z does not occur in N and θ matches the type of M. $\mathsf{let}\, x = M\, \mathsf{in}\, N$ will stand for $(\lambda x^{\theta}.N)M$ in general. The operational semantics of the language can be found, e.g. in [18]. Note that, if $max > 0$, reference equality is expressible in our syntax [19].

Definition 1. *We say that the term-in-context $\Gamma \vdash M_1 : \theta$* **approximates** $\Gamma \vdash M_2 : \theta$ *(written $\Gamma \vdash M_1 \mathrel{\underset{\sim}{\sqsubset}} M_2$) if $C[M_1] \Downarrow$ implies $C[M_2] \Downarrow$ for any context $C[-]$ such that $\vdash C[M_1], C[M_2] : \mathsf{unit}$. Two terms-in-context are* **equivalent** *if one approximates the other (written $\Gamma \vdash M_1 \cong M_2$).*

3 Game semantics

Game semantics views computation as a dialogue between the environment (Opponent, O) and the program (Proponent, P). We give an overview of the fully abstract game model of GRef [14,18]. Let $\mathbb{A} = \biguplus_{\gamma} \mathbb{A}_{\gamma}$ be a collection of countably infinite sets of *reference names*, or just *names*. The model is constructed using mathematical objects (moves, plays, strategies) that will feature names drawn from $\mathbb{A}$. Although names underpin various elements of our model, their precise nature is irrelevant. Hence, all of our definitions preserve name-invariance, i.e. our objects are (strong) *nominal sets* [8, 22]. Note that we do not need the full power of the theory but mainly the basic notion of name-permutation. For an element x belonging to a (nominal) set X, we write $\nu(x)$ for its name-support, i.e. the set of names occurring in x. Moreover, for any $x, y \in X$, we write $x \sim y$ if x and y are the same up to a permutation of $\mathbb{A}$. Our model is couched in the Honda-Yoshida style of modelling call-by-value computation [9]. Before we define what it means to play our games, let us introduce the auxiliary concept of an arena.

Definition 2. *An* arena *$A = \langle M_A, I_A, \lambda_A, \vdash_A \rangle$ is given by a set M_A of moves, its subset I_A of initial ones, a labelling function $\lambda_A : M_A \to \{O, P\} \times \{Q, A\}$ and a justification relation $\vdash_A \subseteq M_A \times (M_A \setminus I_A)$.*
In addition, for all $m, m' \in M_A$, we stipulate: $m \in I_A \implies \lambda_A(m) = (P, A)$, $m \vdash_A m' \wedge \lambda_A^{QA}(m) = A \implies \lambda_A^{QA}(m') = Q$, $m \vdash_A m' \implies \lambda_A^{OP}(m) \neq \lambda_A^{OP}(m')$. We write λ_A^{OP} (resp. λ_A^{QA}) for λ_A post-composed with the first (second) projection.

We shall use ι to range over initial moves. Let $\overline{\lambda_A}$ be the OP-complement of λ_A. Given arenas A, B, the arenas $A \otimes B$ and $A \Rightarrow B$ are constructed as in the following figure,

$$\begin{array}{ll}
M_{A\Rightarrow B} = \{\star\} \uplus M_A \uplus M_B,\ I_{A\Rightarrow B} = \{\star\} & M_{A\otimes B} = (I_A \times I_B) \uplus \bar{I}_A \uplus \bar{I}_B,\ I_{A\otimes B} = I_A \times I_B \\
\lambda_{A\Rightarrow B} = [\star \mapsto PA,\ \overline{\lambda}_A[\iota_A \mapsto OQ],\ \lambda_B] & \lambda_{A\otimes B} = [(\iota_A,\iota_B) \mapsto PA,\ \lambda_A \upharpoonright \bar{I}_A,\ \lambda_B \upharpoonright \bar{I}_B] \\
\vdash_{A\Rightarrow B} = \{(\star,\iota_A),(\iota_A,\iota_B)\} \cup \vdash_A \cup \vdash_B & \vdash_{A\otimes B} = \{((\iota_A,\iota_B),m) \mid \iota_A \vdash_A m \vee \iota_B \vdash_B m\} \\
 & \qquad \cup \bar{\vdash}_A \cup \bar{\vdash}_B
\end{array}$$

where $\bar{I}_A = M_A \setminus I_A, \bar{\vdash}_A = (\vdash_A \upharpoonright \bar{I}_A \times \bar{I}_A)$ (and similarly for B). Let us write $[i,j]$ for the set $\{i, i{+}1, \cdots, j\}$. For each type θ we can define the corresponding arena $[\![\theta]\!]$.

$$\begin{array}{ll}
[\![\mathsf{unit}]\!] = \langle \{\star\},\{\star\},\{(\star,PA)\},\emptyset\rangle & [\![\mathsf{ref}\ \gamma]\!] = \langle \mathbb{A}_\gamma, \mathbb{A}_\gamma, \{(a,PA) \mid a \in \mathbb{A}_\gamma\},\emptyset\rangle \\
[\![\mathsf{int}]\!] = \langle [0,max],[0,max],\{(i,PA) \mid i \in [0,max]\},\emptyset\rangle & [\![\theta \to \theta']\!] = [\![\theta]\!] \Rightarrow [\![\theta']\!]
\end{array}$$

Although types are interpreted by arenas, the actual games will be played in *prearenas*, which are defined in the same way as arenas with the exception that initial moves are O-questions. Given arenas A, B we define the prearena $A \to B$ as follows.

$$\begin{array}{ll}
M_{A\to B} = M_A \uplus M_B & \lambda_{A\to B} = [\overline{\lambda}_A[\iota_A \mapsto OQ], \lambda_B] \\
I_{A\to B} = I_A & \vdash_{A\to B} = \{(\iota_A,\iota_B)\} \cup \vdash_A\ \cup \vdash_B
\end{array}$$

A ***store*** is a type-sensitive finite partial function $\Sigma : \mathbb{A} \rightharpoonup [0,max] \cup \mathbb{A}$ such that $a \in \mathsf{dom}(\Sigma) \cap \mathbb{A}_{\mathsf{int}}$ implies $\Sigma(a) \in [0,max]$, and $a \in \mathsf{dom}(\Sigma) \cap \mathbb{A}_{\mathsf{ref}\,\gamma}$ implies $\Sigma(a) \in \mathsf{dom}(\Sigma) \cap \mathbb{A}_\gamma$. We write Sto for the set of all stores. A move-with-store on a (pre)arena A is a pair m^Σ with $m \in M_A$ and $\Sigma \in \mathsf{Sto}$.

Definition 3. *A* justified sequence *on a prearena A is a sequence of moves-with-store on A such that, apart from the first move, which must be of the form ι^Σ with $\iota \in I_A$, every move $n^{\Sigma'}$ in s is equipped with a pointer to an earlier move m^Σ such that $m \vdash_A n$. m is then called the justifier of n.*

For each $S \subseteq \mathbb{A}$ and Σ we define $\Sigma^0(S) = S$ and $\Sigma^{i+1}(S) = \Sigma(\Sigma^i(S)) \cap \mathbb{A}$ $(i \geq 0)$. Let $\Sigma^*(S) = \bigcup_i \Sigma^i(S)$. The set of *available names* of a justified sequence is defined inductively by $\mathsf{Av}(\epsilon) = \emptyset$ and $\mathsf{Av}(sm^\Sigma) = \Sigma^*(\mathsf{Av}(s) \cup \nu(m))$. The view of a justified sequence is defined by: $view(\epsilon) = \epsilon$, $view(m^\Sigma) = m^\Sigma$ and $view(s\ \overparen{m^\Sigma\ t\ n^{\Sigma'}}) = view(s)\, m^\Sigma n^{\Sigma'}$. We shall write $s \sqsubseteq s'$ to mean that s is a prefix of s'.

Definition 4. *Let A be a prearena. A justified sequence s on A is called a* **play***, if it satisfies the conditions below.*

- *No adjacent moves belong to the same player (*Alternation*).*
- *The justifier of each answer is the most recent unanswered question (*Bracketing*).*
- *For any $s'm^\Sigma \sqsubseteq s$ with non-empty s', the justifier of m occurs in $view(s')$ (*Visibility*).*
- *For any $s'm^\Sigma \sqsubseteq s$, $\mathsf{dom}(\Sigma) = \mathsf{Av}(s'm^\Sigma)$ (*Frugality*).*

Definition 5. *A* **strategy** *σ on a prearena A, written $\sigma : A$, is a set of even-length plays of A satisfying:*

- *If $so^\Sigma p^{\Sigma'} \in \sigma$ then $s \in \sigma$ (*Even-prefix closure*).*
- *If $s \in \sigma$ and $s \sim t$ then $t \in \sigma$ (*Equivariance*).*
- *If $s_1 p_1^{\Sigma_1}, s_2 p_2^{\Sigma_2} \in \sigma$ and $s_1 \sim s_2$ then $s_1 p_1^{\Sigma_1} \sim s_2 p_2^{\Sigma_2}$ (*Nominal determinacy*).*

Following [14,18], GRef-terms $\Gamma \vdash M : \theta$, where $\Gamma = \{x_1 : \theta_1, \cdots, x_n : \theta_n\}$ can be interpreted by strategies for the prearena $[\![\theta_1]\!] \otimes \cdots \otimes [\![\theta_n]\!] \to [\![\theta]\!]$, which we shall denote by $[\![\Gamma \vdash \theta]\!]$. Given a set of plays X, let us write $\mathsf{comp}(X)$ for the set of complete plays in X, i.e. those in which each occurrence of a question justifies an answer. The interpretation given in [14,18] is then fully abstract in the following sense.

Proposition 6 ([14,18]). *Let $\Gamma \vdash M_1, M_2 : \theta$ be* GRef*-terms. $\Gamma \vdash M_1 \sqsubseteq\!\!\!\sim M_2$ if, and only if,* $\mathsf{comp}([\![\Gamma \vdash M_1 : \theta]\!]) \subseteq \mathsf{comp}([\![\Gamma \vdash M_2 : \theta]\!])$*. Hence, $\Gamma \vdash M_1 \cong M_2$ if, and only if,* $\mathsf{comp}([\![\Gamma \vdash M_1 : \theta]\!]) = \mathsf{comp}([\![\Gamma \vdash M_2 : \theta]\!])$.

We shall rely on the result for proving both undecidability and decidability results, by referring to complete plays generated by terms.

Example 7. The name-generating term $\vdash \lambda x^{\mathsf{unit}}.\mathsf{ref}(0) : \mathsf{unit} \to \mathsf{ref\ int}$ yields complete plays of the shape given below (the corresponding prearena is given on the right).

$$\begin{tikzcd}[column sep=small] q & O \\ \star \arrow[u, dash] & P \\ q_0 \arrow[u, dash] & O \\ a \arrow[u, dash] & P \end{tikzcd}$$

$$\begin{tikzcd}[column sep=small] q^{\emptyset} & \star^{\emptyset} \arrow[l, bend right] & q_0^{\Sigma_0'} \arrow[l, bend right] & a_0^{\Sigma_1} \arrow[l, bend right] & \cdots & q_0^{\Sigma_{i-1}'} \arrow[llll, bend right] & a_i^{\Sigma_i} \arrow[l, bend right] & q_0^{\Sigma_i'} \arrow[llllll, bend right] & \cdots \end{tikzcd}$$

where $\Sigma_0' = \emptyset$ and, for all $i > 0$, $\Sigma_i = \Sigma_{i-1}' \cup \{(a_i, 0)\}$, $\mathsf{dom}(\Sigma_i') = \mathsf{dom}(\Sigma_i)$. Moreover, for any $i \neq j$ we have $a_i \neq a_j$. Note that Σ_i' can be different from Σ_i, i.e. the environment is free to change the values stored at all of the locations that have been revealed to it.

Note that in the above example the sizes of stores keep on growing indefinitely. However, the essence of the strategy is already captured by plays of the shape $q \star q_0\, a_0^{(a_0,0)} \cdots q_0\, a_i^{(a_i,0)} q_0 \cdots$ under the assumption that, whenever a value is missing from the store of an O-move, it is arbitrary and, for P-moves, it is the same as in the preceding O-move. Next we spell out how a sequence of moves-with-store, not containing enough information to qualify as a play, can be taken to represent proper plays.

Definition 8. *Let $s = m_1^{\Sigma_1} \cdots m_k^{\Sigma_k}$ be a play over $\Gamma \vdash \theta$ and $t = m_1^{\Theta_1} \cdots m_k^{\Theta_k}$ be a sequence of moves-with-store. We say that s is an* extension *of t if $\Theta_i \subseteq \Sigma_i$ $(1 \leq i \leq k)$ and, for any $1 \leq i \leq \lfloor k/2 \rfloor$, if $a \in \mathsf{dom}(\Sigma_{2i}) \setminus \mathsf{dom}(\Theta_{2i})$ then $\Sigma_{2i}(a) = \Sigma_{2i-1}(a)$. We write $\mathsf{ext}(t)$ for the set of all extensions of t.*

Because we cannot hope to encode plays with unbounded stores through automata, our decidability results will be based on representations of plays that capture strategies via extensions.

4 Undecidability arguments

We begin with undecidable cases. Our argument will rely on queue machines, which are finite-state devices equipped with a queue.

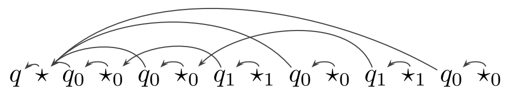
Definition 9. *Let $\mathcal{A}$ be a finite alphabet. A queue machine over $\mathcal{A}$ is specified by $\langle\, Q, Q_E, Q_D, \mathit{init}, \delta_E, \delta_D \,\rangle$, where Q is a finite set of states such that $Q = Q_E \uplus Q_D$, $\mathit{init} \in Q_E$ is the initial state, $\delta_E : Q_E \to Q \times \mathcal{A}$ is the enqueuing function, whereas $\delta_D : Q_D \times \mathcal{A} \to Q$ is the dequeuing function.*

A queue machine starts at state $init$ with an empty queue. Whenever it reaches a state $q \in Q_E$, it will progress to the state $\pi_1\delta_E(q)$ and $\pi_2\delta_E(q)$ will be added to the associated queue. If the machine reaches a state $q \in Q_D$ and its queue is empty, the machine is said to *halt*. Otherwise, it moves to the state $\delta_D(q,x)$, where x is the symbol at the head of the associated queue, which is then removed from the queue. The halting problem for queue machines is well known to be undecidable (e.g. [12]). By encoding computation histories of queue machines as plays generated by GRef terms we next show that the equivalence problem for GRef terms must be undecidable. Note that this entails undecidability of the associated notion of term approximation.

Theorem 10. *The contextual equivalence problem is undecidable in the following cases.*
- $\vdash \mathsf{unit} \to \mathsf{unit} \to \mathsf{unit}$
- $(\mathsf{unit} \to \mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit} \vdash \mathsf{unit}$
- $(((\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \to \mathsf{unit}) \to \mathsf{unit} \vdash \mathsf{unit}$
- $\vdash ((\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \to \mathsf{unit}$

We sketch the argument in the first case. The arena used to interpret closed terms of type unit $\to$ unit $\to$ unit has the shape given on the right. We are going to use plays from the arena to represent sequences of queue operations. Enqueuing will be represented by segments of the form $q_0\star_0$, whereas $q_1\star_1$ will be used to represent dequeuing. Additionally, in the latter case q_1 will be justified by $\star_0$ belonging to the segment representing the enqueuing of the element that is now being dequeued. For instance, the sequence $EEDEDE$, in which E, D stand for enqueing and dequeing respectively, will be represented as follows.

$$\begin{tikzcd}[row sep=small] q \arrow[d, dash] \\ \star \arrow[d, dash] \\ q_0 \arrow[d, dash] \\ \star_0 \arrow[d, dash] \\ q_1 \arrow[d, dash] \\ \star_1 \end{tikzcd}$$


$q\ \star\ q_0\ \star_0\ q_0\ \star_0\ q_1\ \star_1\ q_0\ \star_0\ q_1\ \star_1\ q_0\ \star_0$

Observe that all such plays are complete. Given a queue machine $\mathbb{Q}$, let us write $\mathsf{hist}(\mathbb{Q})$ for the (prefix-closed) subset of $(E \uplus D)^*$ corresponding to all sequences of queue operations performed by $\mathbb{Q}$. Note that $\mathsf{hist}(\mathbb{Q})$ is finite if and only if $\mathbb{Q}$ halts. Additionally, define $\mathsf{hist}^-(\mathbb{Q})$ to be $\mathsf{hist}(\mathbb{Q})$ from which the longest sequence is removed (if $\mathsf{hist}(\mathbb{Q})$ is infinite and the sequence in question does not exist we set $\mathsf{hist}^-(\mathbb{Q}) = \mathsf{hist}(\mathbb{Q})$). Note that the sequence corresponds to a terminating run and necessarily ends in D.

Lemma 11. *Let* $\mathbb{Q}$ *be a queue machine. There exist terms* $\vdash M, M^-\colon \mathsf{unit} \to \mathsf{unit} \to \mathsf{unit}$ *of* GRef *such that* $\mathsf{comp}(\llbracket M \rrbracket)$, $\mathsf{comp}(\llbracket M^- \rrbracket)$ *represent* $\mathsf{hist}(\mathbb{Q})$, $\mathsf{hist}^-(\mathbb{Q})$ *respectively.*

Proof. W.l.o.g. we shall assume that Q can be fitted into int (otherwise, we could use a fixed number of variables to achieve the desired storage capacity). Let $D[-] \equiv C[\lambda x.C_0[\lambda y.C_1[-]]]$, where $C[-], C_0[-], C_1[-]$ are given in Figure 2 ($*$ is a special symbol not in the queue alphabet and Ω is a canonical divergent term). $C_0[-]$ and $C_1[-]$ handle enqueuing and dequeuing respectively. We take M, M^- to be $D[()]$, $D[\mathsf{if}\ (!STATE \in Q_D \wedge !!LAST = *)\ \mathsf{then}\ \Omega]$ respectively. □

Observe that $\mathsf{hist}(\mathbb{Q}) = \mathsf{hist}^-(\mathbb{Q})$ exactly when $\mathbb{Q}$ does not halt. Consequently, the problem of deciding $\mathsf{hist}(\mathbb{Q}) = \mathsf{hist}^-(\mathbb{Q})$ is undecidable. Thus, via Proposition 6, we can conclude that program equivalence is undecidable for closed terms of type unit $\to$ unit $\to$ unit. The remaining cases are treated in a similar manner.

$$\begin{aligned}
C[-] = {} & \mathsf{let}\, STATE = \mathsf{ref}(init)\ \mathsf{in} \\
& \mathsf{let}\, LAST = \mathsf{ref}(\mathsf{ref}(*))\ \mathsf{in}\ [-]
\end{aligned}
\qquad
\begin{aligned}
C_0[-] = {} & \mathsf{if}\ (!STATE \notin Q_E)\ \mathsf{then}\ \Omega; \\
& \mathsf{let}\, SYM = \mathsf{ref}(\pi_2\delta_E(!STATE))\ \mathsf{in} \\
& \mathsf{let}\, PREV = \mathsf{ref}(!LAST)\ \mathsf{in} \\
& \quad STATE := \pi_1\delta_E(!STATE); \\
& \quad LAST := SYM;\ [-]
\end{aligned}$$

$$\begin{aligned}
C_1[-] = {} & \mathsf{if}\ (!STATE \notin Q_D)\ \mathsf{then}\ \Omega; \\
& \mathsf{if}\ (!!PREV \neq * \vee\ !SYM = *)\ \mathsf{then}\ \Omega; \\
& STATE := \delta_D(!STATE, !SYM); \\
& SYM := *;\ [-]
\end{aligned}$$

Fig. 2. Simulating a queue machine in $\vdash \mathsf{unit} \to \mathsf{unit} \to \mathsf{unit}$. The variable $STATE : \mathsf{ref\,int}$ contains the current state of the machine. The queue is encoded as a backwards-connected list with elements $(PREV, SYM) : \mathsf{ref}^2\,\mathsf{int} \times \mathsf{ref\,int}$, with last-element pointer $LAST : \mathsf{ref}^2\,\mathsf{int}$. Enqueuing adds a new last element while dequeuing sets the first non-$*$ symbol of the list to $*$.

5 Decidability

We now focus on a fragment of GRef, called GRef☺, that comprises all types that do *not* fall under the undecidable cases identified earlier.

Definition 12. *Suppose $\Gamma = x_1 : \theta_1, \cdots, x_m : \theta_m$. The term-in-context $\Gamma \vdash M : \theta$ belongs to* GRef☺ *provided $\theta_1, \cdots, \theta_m$ can be generated from Θ_L and θ is generated from Θ_R, where $\Theta_L ::= \beta \mid \Theta_R \to \Theta_L$, $\Theta_R ::= \beta \mid \Theta_1 \to \beta$ and $\Theta_1 ::= \beta \mid \beta \to \Theta_1$.*

Put otherwise, we focus on sequents of the form: $\Theta_R \to \cdots \to \Theta_R \to \beta \vdash \Theta_R$, where $\Theta_R = (\beta \to \cdots \to \beta) \to \beta$. In order to show decidability we first translate GRef☺ terms into automata that represent their game semantics. Any GRef☺ term can be effectively converted to an equivalent term in canonical shape, which is captured by the grammar below. Consequently, it suffices to show that program equivalence between terms in canonical form is decidable. Accordingly, in what follows, we focus exclusively on translating terms in canonical shape.

$$\begin{aligned}
\mathsf{C} ::= {} & () \mid i \mid x^{\mathsf{ref}\,\gamma} \mid \lambda x^{\Theta_1}.\mathsf{C} \mid \mathsf{case}(x^{\mathsf{int}})[\mathsf{C}, \cdots, \mathsf{C}] \mid (\mathsf{while}\,(!x^{\mathsf{ref\,int}})\,\mathsf{do}\,\mathsf{C});\mathsf{C} \\
& \mid \mathsf{let}\,y^{\gamma} = !x^{\mathsf{ref}\,\gamma}\ \mathsf{in}\ \mathsf{C} \mid (x^{\mathsf{ref\,int}} := i);\mathsf{C} \mid (x^{\mathsf{ref}^2\,\gamma} := y^{\mathsf{ref}\,\gamma});\mathsf{C} \\
& \mid \mathsf{let}\,x^{\mathsf{ref\,int}} = \mathsf{ref}(0)\ \mathsf{in}\ \mathsf{C} \mid \mathsf{let}\,x^{\mathsf{ref}^2\,\gamma} = \mathsf{ref}(y^{\mathsf{ref}\,\gamma})\ \mathsf{in}\ \mathsf{C} \mid \mathsf{let}\,y^{\Theta_L} = z\,()\ \mathsf{in}\ \mathsf{C} \\
& \mid \mathsf{let}\,y^{\Theta_L} = z\,i\ \mathsf{in}\ \mathsf{C} \mid \mathsf{let}\,y^{\Theta_L} = z\,x^{\mathsf{ref}\,\gamma}\ \mathsf{in}\ \mathsf{C} \mid \mathsf{let}\,y^{\Theta_L} = z\,(\lambda x^{\Theta_1}.\mathsf{C})\ \mathsf{in}\ \mathsf{C}
\end{aligned}$$

Each type θ can be written in the form $\theta = \theta_n \to \ldots \to \theta_1 \to \beta$, for types $\theta_1, \ldots, \theta_n$ and base type β. For brevity, we shall write $\theta = (\theta_n, \ldots, \theta_1, \beta)$. We call n the *arity* of θ and denote it by $ar(\theta)$. Next we fix notation for referring to moves that are available in arenas corresponding to GRef☺ typing judgments: each move can be viewed as a pair (l, t) subject to consistency constraints induced by the subtypes which contribute them, e.g. the label corresponding to a tag related to int must be a number from $[0, max]$.

Definition 13. *For every type θ let us define the associated set of labels $\mathcal{L}_\theta$ as follows: $\mathcal{L}_{\mathsf{unit}} = \{\star\}$, $\mathcal{L}_{\mathsf{int}} = \{0, \cdots, max\}$, $\mathcal{L}_{\mathsf{ref}\,\gamma} = \mathbb{A}_\gamma$, $\mathcal{L}_{\theta\to\theta'} = \{\star\}$.*
We shall write $\mathcal{L}$ for the set of all labels. Given a GRef☺ *typing judgement $\Gamma \vdash M : \theta$ we write $\mathbb{T}$ for the set of associated* tags*:*

$$\begin{aligned}
\mathbb{T} = {} & \{\mathsf{c}_i, \mathsf{r}_i \mid \theta \equiv \theta' \to \beta,\ 0 \le i \le ar(\theta')\} \cup \{\mathsf{c}_i^x, \mathsf{r}_i^x \mid (x : (\theta_m, \cdots, \theta_1, \beta)) \in \Gamma,\ 0 < i \le m\} \\
& \cup \{\mathsf{c}_{j,i}^x, \mathsf{r}_{j,i}^x \mid (x : (\theta_m, \cdots, \theta_1, \beta)) \in \Gamma,\ 0 < j \le m,\ \theta_j \equiv \theta' \to \beta,\ 0 \le i \le ar(\theta')\} \cup \{\mathsf{r}_\downarrow\}
\end{aligned}$$

partitioned as $\mathbb{T} = \mathbb{T}_{\mathsf{push}} \uplus \mathbb{T}_{\mathsf{pop}} \uplus \mathbb{T}_{\mathsf{noop}}$, *where* $\mathbb{T}_{\mathsf{push}} = \{\mathsf{c}_i, \mathsf{c}_i^x, \mathsf{c}_{j,i}^x \mid i > 0\}$, $\mathbb{T}_{\mathsf{pop}} = \{\mathsf{r}_i, \mathsf{r}_i^x, \mathsf{r}_{j,i}^x \mid i > 0\}$ *and* $\mathbb{T}_{\mathsf{noop}} = \{\mathsf{c}_0, \mathsf{r}_0, \mathsf{c}_{j,0}^x, \mathsf{r}_{j,0}^x\}$.

The automata we shall rely on are equipped with finitely many, say n, registers for storing elements of $\mathbb{A}$, the first n_{r} of which are read-only. The content of registers, called *register assignment*, will be described by an injective partial function $\rho : [1,n] \rightharpoonup \mathbb{A}$. The set of all register assignments will be denoted by Reg and we shall use ρ to range over them. The automata will read elements of $\mathcal{L} \times \mathbb{T} \times \mathsf{Sto}$ (corresponding to moves-with-store) in a single transition step. In order to specify what label is to be read in a given step we use *symbolic labels* from the set $\mathbb{L} = \{\star\} \cup [0, max] \cup \{\, \mathtt{R}_i \mid 1 \leq i \leq n \,\}$ ($\mathtt{R}_i$ stands for the name stored in the ith register). To designate which tag is to be processed we simply use elements of (the finite set) $\mathbb{T}$. To describe stores, we shall use *symbolic stores* from the set $\mathsf{SSto} = \{S : [1,n] \rightharpoonup [0, max] \cup \{\mathtt{R}_1, \cdots, \mathtt{R}_n\} \mid [1, n_{\mathsf{r}}] \subseteq \mathsf{dom}(S)\}$. Symbolic stores represent stores by use of indices instead of actual names (for example, $S(i) = \mathtt{R}_j$ means that in S the i-th name stores the j-th name). Altogether, in order to define our automata, we will use transition labels from the set

$$\mathsf{TL} = \mathcal{P}([n_{\mathsf{r}}{+}1, n]) \times \mathbb{L} \times (((\mathbb{T}_{\mathsf{push}} \uplus \mathbb{T}_{\mathsf{pop}}) \times \mathbb{C}_{\mathsf{stack}} \times \mathsf{Mix}) \uplus \mathbb{T}_{\mathsf{noop}}) \times \mathsf{SSto}$$

where Mix is the set of partial injections $\pi : [n_{\mathsf{r}}{+}1, n] \to [n_{\mathsf{r}}{+}1, n]$ and $\mathcal{P}$ is powerset. Depending on tags involved, the above set can be partitioned into $\mathsf{TL}_{\mathsf{push}}$, $\mathsf{TL}_{\mathsf{pop}}$, $\mathsf{TL}_{\mathsf{noop}}$ respectively. The first component $X \in \mathcal{P}([n_{\mathsf{r}}{+}1, n])$ of each transition label is responsible for name generation: $|X|$ fresh names are to be generated and placed in registers given in X. When the tag corresponds to an O-move (tags $c_0, r_i, r_i^x, r_{j,i}^x, c_{j,0}^x$ $(i > 0)$) freshness is meant to be interpreted locally, i.e. none of the new names can be present in the current register. For P-moves (tags $\mathsf{r}_\downarrow, r_0, c_i, c_i^x, c_{j,i}^x, r_{j,0}^x$ $(i > 0)$), we require *global* freshness, i.e. that the names have not been encountered before by the automaton.

The $((\mathbb{T}_{\mathsf{push}} \uplus \mathbb{T}_{\mathsf{pop}}) \times \mathbb{C}_{\mathsf{stack}} \times \mathsf{Mix}) \uplus \mathbb{T}_{\mathsf{noop}}$ part corresponds to stack actions. Our automata will use a visibly pushdown stack [4], where the tags determine stack actions according to the partition into $\mathbb{T}_{\mathsf{push}}$, $\mathbb{T}_{\mathsf{pop}}$ and $\mathbb{T}_{\mathsf{noop}}$. The stack will be used to store elements from $\mathbb{C}_{\mathsf{stack}} \times \mathsf{Reg}$, where $\mathbb{C}_{\mathsf{stack}}$ is a finite set of stack symbols (thus, on the stack we will store stack symbols together with register assignments). Note that in transition labels the tags from $\mathbb{T}_{\mathsf{push}} \uplus \mathbb{T}_{\mathsf{pop}}$ come with $(s, \pi) \in \mathbb{C}_{\mathsf{stack}} \times \mathsf{Mix}$. For push-tags, s, π indicate that s should be pushed along with register assignment $\rho \circ \pi$, where ρ is the present content of the registers, i.e. we only push the (content of) registers from $\mathsf{cod}(\pi)$ reindexed according to π. For pop-tags, s, π indicate what stack symbol should occur on top of the stack and π spells out the expected relationship between the present register assignment ρ and the assignment ρ' on top of the stack: for popping to take place we require $\rho(i) = \rho'(j)$ iff $(i, j) \in \pi$. The content of registers in $\mathsf{dom}(\rho') \setminus \mathsf{cod}(\pi)$ will then be popped directly into the machine registers without reindexing.

Name-generation is meant to occur before pushing (so the new names can end up on the stack as soon after being generated), but after popping. The symbolic store $S \in \mathsf{SSto}$ in a transition label is also interpreted after name generation (so that fresh names can occur in stores). Assuming ρ is the register assignment obtained after popping and name generation, S stipulates that the move which is being read must come with the store $\Sigma = \{\, (\rho(i), S(i)) \mid S(i) \in [0, max] \,\} \cup \{\, (\rho(i), \rho(j)) \mid S(i) = \mathtt{R}_j \,\}$ (this definition will be valid because we shall always have $\mathsf{dom}(\rho) = \mathsf{dom}(S)$). Formally, the automata we use are defined as follows.

Definition 14. *An* (n_{r}, n)**-automaton** *of type* θ *is given as* $\mathcal{A} = \langle Q, q_0, \rho_0, \delta, F\rangle$ *where:*

- Q *is a finite set of states, partitioned into* Q_O *(O-states) and* Q_P *(P-states);*
- $q_0 \in Q_P$ *is the initial state;* $F \subseteq Q_O$ *is the set of final states;*
- $\rho_0 \in \mathsf{Reg}$ *is the initial register assignment such that* $[1, n_{\mathsf{r}}] \subseteq \mathsf{dom}(\rho_0)$*;*
- $\delta \subseteq (Q_P \times (\mathsf{TL}_{\mathsf{push}} \cup \mathsf{TL}_{\mathsf{noop}}) \times Q_O) \cup (Q_O \times (\mathsf{TL}_{\mathsf{pop}} \cup \mathsf{TL}_{\mathsf{noop}}) \times Q_P) \cup (Q_O \times \mathsf{Mix} \times Q_O) \cup (Q_P \times \mathsf{Mix} \times Q_P)$ *is the transition relation.*

Additionally, if θ *is a base type then there is a unique final state* q_F*, while* $\delta \upharpoonright \{q_F\} = \emptyset$ *(no outgoing transitions) and* $\delta^{-1} \upharpoonright \{q_F\} \subseteq \{q_F\} \times \mathsf{TL}_{\mathsf{noop}} \times Q_P$ *(reach only by no-op).*

Note that in addition to the labelled transitions discussed earlier, we allow ϵ-transitions $(q_1, \pi, q_2) \in \delta$ which rearrange the contents of registers in $[n_{\mathsf{r}}{+}1, n]$ according to $\pi \in \mathsf{Mix}$: if the automaton is in state q_1 and the current register assignment is ρ, after the transition the automaton will move to q_2 and the new register assignment will be $\rho \circ \overline{\pi}$, where $\overline{\pi}(i) = i$ $(1 \leq i \leq n_{\mathsf{r}})$ and $\overline{\pi}(i) = \pi(i)$ (otherwise). We write $L(\mathcal{A})$ for the set of words from $(\mathcal{L} \times \mathbb{T} \times \mathsf{Sto})^*$ that are accepted by $\mathcal{A}$ by final state.

Given $\Gamma = \{x_1 : \theta_1, \cdots, x_m : \theta_m\}$ and θ, let us write $P^1_{\Gamma\vdash\theta}$ for the set of plays of length 1 over $[\![\Gamma \vdash \theta]\!]$. Recall that each of them will have the form ι^{Σ_0}, where $\iota \in I_\Gamma$, i.e. $\iota = (l_1, \cdots, l_m)$ with $l_i \in \mathcal{L}_{\theta_i}$. Moreover, the names in ι^{Σ_0} coincide with those of $\mathsf{dom}(\Sigma_0) = \nu(\Sigma_0)$. We order them by use of register assignments and set:

$$I^+_{\Gamma\vdash\theta} = \{(\iota^{\Sigma_0}, \rho_0) \mid \iota^{\Sigma_0} \in P^1_{\Gamma\vdash\theta},\, \nu(\rho_0) = \nu(\Sigma_0),\, \exists k.\, \rho_0([1,k]) = \nu(\iota)\}$$

For brevity, we shall write each element $(\iota^{\Sigma_0}, \rho_0) \in I^+_{\Gamma\vdash\theta}$ as $\iota^{\Sigma_0}_{\rho_0}$.

Lemma 15. *Let* $\Gamma \vdash \mathsf{C} : \theta$ *be a* $\mathsf{GRef}☺$*-term in canonical form. For each* $j = \iota^{\Sigma_0}_{\rho_0} \in I^+_{\Gamma\vdash\theta}$*, there exists a deterministic* $(|\nu(\iota)|, m_j)$*-automaton* $\mathcal{A}_j$ *with initial register assignment* ρ_0 *such that* $\bigcup_{w\in L(\mathcal{A}_j)} \mathsf{ext}(\iota^{\Sigma_0} w) = \mathsf{comp}([\![\Gamma \vdash \mathsf{C} : \theta]\!]) \cap P^{\iota^{\Sigma_0}}_{\Gamma\vdash\theta}$*, where* $P^{\iota^{\Sigma_0}}_{\Gamma\vdash\theta}$ *is the set of plays over* $[\![\Gamma \vdash \theta]\!]$ *that start from* ι^{Σ_0}*.*

Proof. We build upon the techniques developed in [10] insofar as the non-nominal part of the constructions and pointers are concerned. The essence of the nominal approach is revealed in the construction of the automaton, say $\mathcal{A}'$, corresponding to $\mathsf{let}\, x^{\mathsf{ref}^2\,\gamma} = \mathsf{ref}(y^{\mathsf{ref}\,\gamma})\,\mathsf{in}\,\mathsf{C}$. This is done inductively, starting from the automaton for C, call it $\mathcal{A}$, in which x appears in the initial assignment with name a. Passing to $\mathcal{A}'$ then amounts to omitting a from all transitions in $\mathcal{A}$ (along with parts of the store that can only be reached through a) as long as a has not been played in a P-transition (scenarios in which this happens in an O-transition are discarded). At that point, we convert the transition into one which creates a fresh name and then proceed as $\mathcal{A}$. The construction is complicated by the fact that, while a is being omitted from the store, we need to keep track of its value inside the state and the value may be a chain of other hidden names.

As in [10], the hardest construction is that of the automaton for $\mathsf{let}\, y = z(\lambda x.\mathsf{C})\,\mathsf{in}\,\mathsf{C}'$. Because of the structure of the arenas involved, the automaton needs to be designed so that it can alternate between plays in C and C'. More specifically, from designated states in C, we need to allow for jumps to C' and vice versa. In [10], such jumps involve the stack so that the well-bracketing condition is preserved: each call is matched to the appropriate return.[5] In our setting the jumps involve the stack also in a more crucial way:

[5] Jumps to C' are made by P-calls of specific type in C, and returns by corresponding O-returns.

when jumping to C', the automaton stores its register assignment to the stack so that, once control returns to C, the state can be recovered and computation can resume from where it had been interrupted. Such interleaving of computations from C and C' requires frequent rearrangements of registers to make sure the respective register assignments of C and C' are simulated in the single register assignment of the automaton we construct. For this, we follow a two-step construction which involves first introducing a notion of automaton operating on two distinct register assignments that do not interfere with each other, which we next reduce to an (n_{r}, n)-automaton. $\square$

Lemma 16. *Let $\Gamma \vdash \mathsf{C}_1, \mathsf{C}_2 : \theta$ be* GRef☺*-terms in canonical form. For each $j = \iota_{\rho_0}^{\Sigma_0} \in I_{\Gamma\vdash\theta}^{+}$, there exists a deterministic $(|\nu(\iota)|, n_j)$-automaton $\mathcal{B}_j$ with initial register assignment ρ_0 such that $L(\mathcal{B}_j) = \emptyset$ iff* $\mathsf{comp}([\![\Gamma \vdash \mathsf{C}_1 : \theta]\!]) \cap P_{\Gamma\vdash\theta}^{\iota^{\Sigma_0}} \subseteq [\![\Gamma \vdash \mathsf{C}_2 : \theta]\!]$.

Proof. Let $\mathcal{A}_j^1, \mathcal{A}_j^2$ be the automata obtained from the previous Lemma for $\mathsf{C}_1, \mathsf{C}_2$ respectively. Because our automata use visibly pushdown stacks and rely on the same partitioning of tags, we can synchronize them using a single stack and check whether any complete play from $P_{\Gamma\vdash\theta}^{\iota^{\Sigma_0}}$ represented by $\mathcal{A}_j^1$ is also represented by $\mathcal{A}_j^2$. Note that this is not a direct inclusion check, because the automata represent plays via extensions and the representations are not uniquely determined. Consequently, $\mathcal{B}_j$ synchronizes $\mathcal{A}_j^1, \mathcal{A}_j^2$ and also explores each possible way of relating names that are present in stores. Once a clash is detected (e.g. different store values, global freshness vs existing name), $\mathcal{B}_j$ continues to simulate $\mathcal{A}_j^1$ only to see whether the offending scenario extends to a complete play of C_1. If so, it enters an accepting state. $\square$

Note that, although $I_{\Gamma\vdash\theta}^{+}$ is an infinite set, there exists a finite subset $J \subseteq I_{\Gamma\vdash\theta}^{+}$ such that $\{\mathcal{A}_j\}_{j\in J}$ already captures $\mathsf{comp}([\![\Gamma \vdash \mathsf{C} : \theta]\!])$, because up to name-permutation there are only finitely many initial moves. Consequently, we only need finitely many of them to check whether $\Gamma \vdash \mathsf{C}_1 \sqsubseteq_{\sim} \mathsf{C}_2$. By Lemma 16, to achieve this we need to be able to decide the emptiness problem for (n_{r}, n)-automata. To show that it is indeed decidable we consider register pushdown automata [6] over infinite alphabets. They are similar to (n_{r}, n)-automata in that they are equipped with registers and a stack. The only significant differences are that they process single names in a computational step and do not have the ability to generate globally fresh names. The first difficulty is easily overcome by decomposing transitions of our automata into a bounded number of steps (the existence of the bound follows from the fact that symbolic stores in our transition function are bounded). To deal with freshness we need a separate argument.

Lemma 17. *The emptiness problem for register pushdown automata extended with fresh-symbol generation is decidable.*

Proof. Register pushdown automata have the ability to generate (locally fresh) symbols not present in the current registers. However, using them directly as a substitute for global freshness is out of question, because it would lead to spurious accepting runs due to the presence of the stack. To narrow the gap, we observe that if the automata could generate locally fresh symbols that are *in addition* not present on the stack then, for the purposes of emptiness testing, this stronger generative power, which we refer to as *quasi freshness*, could stand in for global freshness. It turns out that a run involving quasi fresh symbols can be simulated through local freshness. Whenever a quasi fresh name is generated, we will generate a locally fresh name annotated with a tag indicating its supposed quasi-freshness. The tags will accompany such names as they are being pushed from registers and popped back into them. Whenever we find that a tagged name in a register coincides with a name on top of the stack, we will interrupt the computation as this will indicate that the supposedly quasi-fresh symbol is not quasi fresh. If no violations are detected through tags, we can show that annotated locally fresh names in an accepting run may well be replaced by quasi-fresh ones. □

Finally, combining Proposition 6 with Lemmata 15-17 we obtain:

Theorem 18. *Program approximation (and thus program equivalence) is decidable for* GRef☺*-terms.*

References

1. S. Abramsky, D. Ghica, A. S. Murawski, C.-H. L. Ong and I. Stark. Nominal games and full abstraction for the nu-calculus. In *LICS*, pp 150–159, 2004.
2. S. Abramsky and G. McCusker. Call-by-value games. In *CSL*, *LNCS* 1414, pp 1–17, 1997.
3. A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *POPL*, pp 340–353, 2009.
4. R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, pp 202–211, 2004.
5. N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *TLCA*, *LNCS* 3461, pp 86–101, 2005.
6. E. Y. C. Cheng and M. Kaminski. Context-free languages over infinite alphabets. *Acta Inf.* 35(3), pp 245–267, 1998.
7. D. Dreyer, Ge. Neis and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *ICFP*, pp 143–156, 2010.
8. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Asp. Comput.* 13, pp 341–363, 2002.
9. K. Honda and N. Yoshida. Game-theoretic analysis of call-by-value computation. *TCS*, 221(1–2):393–456, 1999.
10. D. Hopkins, A. S. Murawski, and C.-H. L. Ong. A fragment of ML decidable by visibly pushdown automata. In *ICALP*, *LNCS* 6756, pp 149–161, 2011.
11. A. Jeffrey and J. Rathke. Towards a theory of bisimulation for local names. In *LICS*, 1999.
12. D. Kozen. *Automata and Computability*. Springer, 1997.
13. V. Koutavas and M. Wand. Small bisimulations for reasoning about higher-order imperative programs In *POPL*, pp 141–152, 2006.
14. J. Laird. A game semantics of names and pointers. *APAL* 151: 151–169, 2008.
15. A. S. Murawski. Functions with local state: regularity and undecidability. *TCS* 338, 2005.
16. A. S. Murawski and N. Tzevelekos. Full abstraction for Reduced ML. In *FOSSACS*, 2009.
17. A. S. Murawski and N. Tzevelekos. Algorithmic nominal game semantics. In *ESOP*, 2011.
18. A. S. Murawski and N. Tzevelekos. Game semantics for good general references. *LICS*, 2011.
19. A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In Gordon and Pitts (eds), *Higher-Order Operational Techniques in Semantics*, pp 227–273. CUP, 1998.
20. U. S. Reddy and H. Yang. Correctness of data representations involving heap data structures. *Sci. Comput. Program.*, 50(1-3):129–160, 2004.
21. D. Sangiorgi, N. Kobayashi and E. Sumii. Environmental bisimulations for higher-order languages. *ACM Trans. Program. Lang. Syst.*, 33(1):5:1–5:69, 2011.
22. N. Tzevelekos. Full abstraction for nominal general references. *LMCS*, 5(3:8), 2009.