ML and Extended Branching VASS

Conrad Cotton-Barratt[1(✉)], Andrzej S. Murawski[2], and C.-H. Luke Ong[1]

[1] University of Oxford, Oxford, UK
conrad.cotton-barratt@cs.ox.ac.uk
[2] University of Warwick, Coventry, UK
a.murawski@warwick.ac.uk

Abstract. We prove that the observational equivalence problem for a finitary fragment of ML is recursively equivalent to the reachability problem for extended branching vector addition systems with states (EBVASS). Our proof uses the fully abstract game semantics of the language. We introduce a new class of automata, VPCMA, as a representation of the game semantics. VPCMA are a version of class memory automata equipped with a visibly pushdown stack; they serve as a bridge enabling interreducibility of decision problems between the game semantics and EBVASS. The results of this paper complete our programme to give an automata classification of the ML types with respect to the observational equivalence problem for closed terms.

1 Introduction

RML is a prototypical call-by-value functional language with state [3], which may be viewed as the canonical restriction of Standard ML to ground-type references. This paper is about the decidability of observational equivalence of finitary RML. Recall that two terms-in-context are *observationally* (or *contextually*) *equivalent*, written $\Gamma \vdash M \cong N$, if they are interchangeable in all program contexts without causing any observable difference in the computational outcome. Observational equivalence is a compelling notion of program equality, but it is hard to reason about because of the universal quantification over program contexts. Our ultimate goal is to completely classify the decidable fragments of finitary RML, and characterise each fragment by an appropriate class of automata. In the case of finitary Idealized Algol [27] – the call-by-name counterpart of RML, the decidability of observational equivalence depends on the type-theoretic order [21] of the terms. By contrast, the decidability of RML terms is not neatly characterised by order: there are undecidable fragments of terms-in-context of order as low as 2 [20], amidst interesting decidable fragments at each of orders 1 to 4. Indeed, as we shall see, there is a pair of second-order types[1] with opposite decidability status but which differs only in the ordering of their argument types.

Let $\mathcal{L}$ be a collection of finitary RML terms-in-context. The observational equivalence problem asks: given two terms-in-context $(i = 1, 2)$

$$x_1 : \theta_1, \cdots, x_k : \theta_k \vdash M_i : \theta$$

[1] Namely, $\mathsf{unit} \to (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$ *vs* $(\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit} \to \mathsf{unit}$.

H. Yang (Ed.): ESOP 2017, LNCS 10201, pp. 314–340, 2017.
DOI: 10.1007/978-3-662-54434-1_12

from $\mathcal{L}$, are they observationally equivalent? Unsurprisingly the general problem is undecidable [20]. However decidability has been established for certain fragments, which we present in Fig. 1 by listing for each fragment the shapes of types allowable on the LHS and RHS of the turnstile, where β is a base type.[2]

Shape	LHS Type, θ_i	RHS Type, θ
I [9]	$(\beta \to \beta) \to \cdots \to (\beta \to \beta) \to \beta$	$\beta \to \cdots \to \beta$
II [15]	$((\beta \to \cdots \to \beta) \to \beta) \to \cdots \to ((\beta \to \cdots \to \beta) \to \beta) \to \beta$	$(\beta \to \cdots \to \beta) \to \beta$

Fig. 1. Two decidable fragments of finitary RML

Note that (the RHS type) θ of shape I ranges over all first-order types; and θ of shape II admits the simplest second-order types. Because [9] also establishes undecidability for the second-order type $\theta = (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit} \to \mathsf{unit}$ and the simplest third-order type $\theta = ((\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \to \mathsf{unit}$, as far as closed terms are concerned, the only unclassified cases are second-order types of the shape

$$\underbrace{\beta \to \cdots \to \beta}_{m} \to (\underbrace{\beta \to \cdots \to \beta}_{n}) \to \beta \tag{1}$$

where $m \geq 1$ and $n \geq 2$. These types are the subject of this paper.

Our main contribution concerns the closed terms of types of the shape

$$\beta \to (\beta \to \cdots \to \beta) \to \beta \tag{2}$$

and relates their observational equivalence problem to the reachability problem for *extended branching vector addition systems with states* (EBVASS) [17], whose decidability status is, to our knowledge, unknown. Our result applies not only to closed terms but also to the fragment $\mathrm{RML}_{\mathrm{EBVASS}}$ (Definition 4) of open terms of type (2) in which free variables are subject to certain type constraints. Our main result is the following

Theorem 1. *Observational equivalence for the terms-in-context in* $\mathrm{RML}_{\mathrm{EBVASS}}$ *is recursively equivalent to the reachability problem for extended branching vector addition systems.*

Our second result (Theorem 23) is that the reachability problem for *reset vector addition systems with states* [5] is reducible to the observational equivalence of closed terms of type $\beta \to \beta \to (\beta \to \beta) \to \beta$. It follows that the observational equivalence of closed terms of all of the remaining types of the shape (1), i.e., where $m, n \geq 2$, is undecidable.

In the following, we discuss the key ideas behind the main results. Like the earlier results [9,15], Theorems 1 and 23 are proved by appealing to the

[2] For the sake of clarity, we do not list types with $\mathsf{int\ ref}$ and the corresponding constraints. They are analogous to treating $\mathsf{int\ ref}$ as $\beta \to \beta$.

game semantics for RML [3,13], which is *fully abstract*, i.e., the equational theory induced by the semantics coincides with observational equivalence. In game semantics [1,16], player P takes the viewpoint of the term-in-context, and player O takes the viewpoint of the program context or environment. Thus a term-in-context, $\Gamma \vdash M : \theta$ with $\Gamma = x_1 : \theta_1, \cdots, x_n : \theta_n$, is interpreted as a P-strategy $[\![\Gamma \vdash M : \theta]\!]$ in the prearena $[\![\theta_1, \cdots, \theta_n \vdash \theta]\!]$. A play is a sequence of moves, made alternately by O and P, such that each non-initial move has a justification pointer to some earlier move. Thanks to the fully abstract game semantics of RML [3,13], observational equivalence is characterised by *complete plays*, i.e., $\Gamma \vdash M \cong N$ holds iff the respective P-strategies, $[\![\Gamma \vdash M : \theta]\!]$ and $[\![\Gamma \vdash N : \theta]\!]$, contain the same set of complete plays. Strategies may be viewed as highly constrained processes, and are amenable to automata-theoretic representations. The main technical challenge, however, lies in the encoding of the justification pointers of the plays.

In recent work [8,9], we considered finitary RML terms-in-context with types of shape I (see Fig. 1). To represent the plays in the game semantics of such terms, we need to encode O-pointers (i.e. justification pointers from O-moves), which is tricky because O-moves are controlled by the environment rather than the term. It turns out that the game semantics of these terms are representable as *nested data class memory automata* (NDCMA) [10], which are a variant of *class memory automata* [6] whose data values exhibit a tree structure, reflecting the tree structure of the threads in the plays.

Because of the type constraints, a play (in the strategy denotation) of a term in $\text{RML}_{\text{EBVASS}}$ may be viewed as an interleaving of "visibly pushdown" threads, subject to the global well-bracketing condition. (See Sect. 3 for an explanation.) In order to model such plays, we introduce *visibly pushdown class memory automata* (VPCMA), which naturally augment class memory automata with a stack and follow a visibly pushdown discipline, but also add data values to the stack so that matching push- and pop-moves must share the same data value. To give a clear representation of the game semantics, we introduce a slight variant of VPCMA with a run-time constraint on the words accepted, called *scoping VPCMA* (SVPCMA). This constraint prevents data values from being read once the stack element that was at the top of the stack when the data value was first read in the run has been popped off the stack. Although these two models are expressively different, they have equivalent emptiness problems.

Unlike in class memory automata (CMA), weakness[3] does not affect the hardness of the emptiness problem for VPCMA, as the stack can be used to check the local acceptance condition. However, like CMA, weakness does help with the closure properties of the languages recognised. The closure properties of these automata are the same as for normal CMA [9]: weak deterministic VPCMA are closed under union, intersection and complementation; similarly for SVPCMA. We show that the complete plays in the game semantics of each $\text{RML}_{\text{EBVASS}}$ term-in-context are representable as a weak deterministic SVPCMA

[3] *Weak* class memory automata [8,9] are class memory automata in which the local acceptance condition is dropped.

(Lemma 14). Thanks to the closure property of SVPCMA, it then follows that $\mathrm{RML}_{\mathrm{EBVASS}}$ observational equivalence is reducible to the emptiness problem for VPCMA (Theorem 13).

Finally and most importantly, we show (Theorems 20 and 22) that the emptiness problem for VPCMA (equivalently for SVPCMA) is equivalent to the reachability problem for extended branching VASS (EBVASS) [17], the decidability of which remains an open problem. In particular, reachability in EBVASS is a harder problem than the long-standing open problem of reachability in BVASS (equivalently, provability in multiplicative exponential linear logic) [11], which is known to be non-elementary [19].

In summary, the results complete our programme to give an automata classification of the ML types with respect to the observational equivalence problem for closed terms of finitary RML. We tabulate our findings as follows:

Order	Type	Automata/status
1	$\mathsf{unit} \to \cdots \to \mathsf{unit}$	NDCMA/decidable [9,14]
2	$(\mathsf{unit} \to \cdots \to \mathsf{unit}) \to \mathsf{unit}$	VPA/decidable [15]
2	$\mathsf{unit} \to (\mathsf{unit} \to \cdots \to \mathsf{unit}) \to \mathsf{unit}$	EBVASS (this paper)
2	$\mathsf{unit} \to \mathsf{unit} \to (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$	Undecidable (this paper)
2	$(\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit} \to \mathsf{unit}$	Undecidable [9]
3	$((\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \to \mathsf{unit}$	Undecidable [9]

Related Work. Hopkins and Murawski [14] used deterministic class memory automata to recognise the strategies of RML terms of a first-order type with certain constraints on the types of their free variables. Building on this idea, strategies of terms-in-context with shape-I types (Fig. 1) are shown to be representable as NDCMA [9]. Automata over an infinite alphabet (specifically, pushdown register automata) have also been applied to game semantics [24,25] for a different purpose, namely, to model generation of fresh names in fragments of ML [25] and Java [22]. When extended with name storage, observational equivalence of terms-in-context with types in $\mathrm{RML}_{\mathrm{EBVASS}}$ becomes undecidable [25]; in particular, this is already the case for closed terms of type $\mathsf{unit} \to \mathsf{unit} \to \mathsf{unit}$.

Outline. In Sect. 2 we define the syntax and operational semantics of RML and the fragment $\mathrm{RML}_{\mathrm{EBVASS}}$. In Sect. 3 we present the game semantics for RML. The automata models, VPMCA and SVPCMA, are then presented in Sect. 4, where we show that their emptiness problems are interreducible, and discuss their closure properties. In Sect. 5 we show that the complete plays in the game semantics of $\mathrm{RML}_{\mathrm{EBVASS}}$-terms are representable as weak deterministic SVPCMA. Consequently the observational equivalence of $\mathrm{RML}_{\mathrm{EBVASS}}$-terms is reducible to the emptiness problem of SVPCMA (and equivalently to that of VPCMA). Reducibility in the opposite direction is then shown in Sect. 6. In Sect. 7 we introduce EBVASS and show that its reachability problem and the

emptiness problem for VPCMA are interreducible. Finally, in Sect. 8, we show that observational equivalence for closed terms of type $\mathsf{unit} \to \mathsf{unit} \to (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$ is undecidable.

2 A Stateful Call-by-Value Functional Language RML

RML is a call-by-value functional language with state [3]. Its types are generated from ground types of int and unit, which represent integers and commands respectively, and the variable type int ref. As the int and unit types will be very similar in their behaviour for our purposes, we will often use β to range over int and unit. Types are then constructed from these in the normal way, using the $\to$ operator:

$$\theta ::= \mathsf{int} \mid \mathsf{unit} \mid \mathsf{int\ ref} \mid \theta \to \theta.$$

The *order* of a type is given by: $\mathbf{ord}(\mathsf{int}) = \mathbf{ord}(\mathsf{unit}) := 0$, $\mathbf{ord}(\mathsf{int\ ref}) := 1$, and $\mathbf{ord}(\theta \to \theta') := max(\mathbf{ord}(\theta) + 1, \mathbf{ord}(\theta'))$. In order to eliminate obvious sources of undecidability, we consider *finitary* RML, with finite ground types ($\mathsf{int} = \{0, \cdots, \max\}$), and iteration instead of recursion. The syntax and typing rules of RML terms are given by induction over the rules in Fig. 2. Note that although we only include arithmetic operations **succ**() and **pred**(), other operations are easily definable using case distinction, because we work with finite int. We will write $\mathbf{let}\, x = M \,\mathbf{in}\, N$ as syntactic sugar for $(\lambda x.N)M$, and $M; N$ for $\mathbf{let}\, x = M \,\mathbf{in}\, N$ where x is chosen to be fresh in N.

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

$$\frac{\Gamma \vdash M : \mathsf{int}}{\Gamma \vdash \mathbf{succ}(M) : \mathsf{int}} \qquad \frac{\Gamma \vdash M : \mathsf{int}}{\Gamma \vdash \mathbf{pred}(M) : \mathsf{int}}$$

$$\frac{\Gamma \vdash M : \mathsf{int} \qquad \Gamma \vdash M_0 : \theta \qquad \Gamma \vdash M_1 : \theta}{\Gamma \vdash \mathbf{if}\, M \,\mathbf{then}\, M_1 \,\mathbf{else}\, M_0 : \theta}$$

$$\frac{\Gamma \vdash M : \mathsf{int\ ref}}{\Gamma \vdash !M : \mathsf{int}} \qquad \frac{\Gamma \vdash M : \mathsf{int\ ref} \qquad \Gamma \vdash N : \mathsf{int}}{\Gamma \vdash M := N : \mathsf{unit}} \qquad \frac{\Gamma \vdash M : \mathsf{int}}{\Gamma \vdash \mathbf{ref}\, M : \mathsf{int\ ref}}$$

$$\frac{\Gamma \vdash M : \mathsf{unit} \to \mathsf{int} \qquad \Gamma \vdash N : \mathsf{int} \to \mathsf{unit}}{\Gamma \vdash \mathbf{mkvar}(M, N) : \mathsf{int\ ref}} \qquad \frac{\Gamma \vdash M : \mathsf{int} \qquad \Gamma \vdash N : \mathsf{unit}}{\Gamma \vdash \mathbf{while}\, M \,\mathbf{do}\, N : \mathsf{unit}}$$

$$\frac{\Gamma \vdash M : \theta \to \theta' \qquad \Gamma \vdash N : \theta}{\Gamma \vdash MN : \theta'} \qquad \frac{\Gamma, x : \theta \vdash M : \theta'}{\Gamma \vdash \lambda x^{\theta}.M : \theta \to \theta'}$$

Fig. 2. Syntax of finitary RML

The operational semantics of the language is presented as a "big-step" relation that uses *stores* [3] to capture the behaviour of variables. Let L range over a

countable set of *locations*, then a store is just a partial function $s : L \to \mathbb{N}_{\leq \max}$. For $l \in L$ and $i \in \{0, \cdots, \max\}$, we write $s[l \mapsto i]$ for the store obtained from s by making l map to i, and for a store s we write $dom(s)$ for the value in L on which s is defined. The reduction rules are defined inductively on pairs (s, M) where s is a store, by the rules presented in Fig. 3. We assume $\max + 1 = \max$ and $0 - 1 = 0$. These reductions reduce terms to *canonical forms*, V, which can be the empty command $()$, a constant integer i, a location l, a lambda-abstraction term $\lambda x.M$, or a *bad-variable construct* using canonical forms inside, $\mathbf{mkvar}(V_1, V_2)$.

$$\frac{}{s, V \Downarrow s, V} \qquad \frac{s, M \Downarrow s', i}{s, \mathbf{succ}(M) \Downarrow s', i+1} \qquad \frac{s, M \Downarrow s', i}{s, \mathbf{pred}(M) \Downarrow s', i-1}$$

$$\frac{s, M \Downarrow s', 0 \qquad s', N_1 \Downarrow s'', V}{s, \mathbf{if}\ M\ \mathbf{then}\ N_0\ \mathbf{else}\ N_1 \Downarrow s'', V} \qquad \frac{s, M \Downarrow s', n+1 \qquad s', N_0 \Downarrow s'', V}{s, \mathbf{if}\ M\ \mathbf{then}\ N_0\ \mathbf{else}\ N_1 \Downarrow s'', V}$$

$$\frac{s, M \Downarrow s', n}{s, \mathbf{ref}\ M \Downarrow s'[l \mapsto n], l}\ l \notin dom(s) \qquad \frac{s, M \Downarrow s', l}{s, !M \Downarrow s', s'(l)}$$

$$\frac{s, M \Downarrow s', l \qquad s', N \Downarrow s'', n}{s, M := N \Downarrow s''[l \mapsto n], ()} \qquad \frac{s, M \Downarrow s', \mathbf{mkvar}(V_0, V_1) \qquad s', V_0() \Downarrow s'', V}{s, !M \Downarrow s'', V}$$

$$\frac{s, M \Downarrow s', \mathbf{mkvar}(V_0, V_1) \qquad s', N \Downarrow s'', n \qquad s'', V_1 n \Downarrow s''', V}{s, M := N \Downarrow s''', V}$$

$$\frac{s, M \Downarrow s', V_1 \qquad s', N \Downarrow s'', V_2}{s, \mathbf{mkvar}(M, N) \Downarrow s'', \mathbf{mkvar}(V_1, V_2)} \qquad \frac{s, M \Downarrow s', 0}{s, \mathbf{while}\ M\ \mathbf{do}\ N \Downarrow s', ()}$$

$$\frac{s, M \Downarrow s', n \qquad s', N \Downarrow s'', () \qquad s'', \mathbf{while}\ M\ \mathbf{do}\ N \Downarrow s''', ()}{s, \mathbf{while}\ M\ \mathbf{do}\ N \Downarrow s''', ()}\ n \neq 0$$

$$\frac{s, M \Downarrow s', \lambda x.M' \qquad s', N \Downarrow s'', V \qquad s'', M'[V/x] \Downarrow s''', V'}{s, MN \Downarrow s''', V'}$$

Fig. 3. Operational semantics of RML

Observational equivalence (OE), also known as contextual equivalence, is the problem of whether two program-fragments are interchangeable without causing any changes to the observable computational outcome. We give a formal definition in Definition 2. OE is a natural notion of program equivalence, a key problem in verification [12].

Definition 2. *Given an* RML *term M, we write $M\Downarrow$ if there exist s and V such that $\emptyset, M \Downarrow s, V$ (where $\emptyset$ is the empty store).*

We say two terms $\Gamma \vdash M : \theta$ and $\Gamma \vdash N : \theta$ are observationally equivalent *if for all contexts $C[-]$ such that $\Gamma \vdash C[M], C[N] : \mathsf{unit}$, $C[M] \Downarrow$ iff $C[N] \Downarrow$.*

Remark 3. RML is similar to Reduced ML [26], the restriction of Standard ML to ground-type references, but is augmented with a "bad-variable" constructor in the sense of Reynolds [27] (in the absence of the constructor, the equality test is definable). In the presence of $\mathsf{int\ ref}$, RML is generally more discriminating then Reduced ML. However observational equivalence of RML coincides with that of Reduced ML on types in which all occurrences (if any) of $\mathsf{int\ ref}$ are positive. The semantics of $\mathsf{int\ ref}$-types in Reduced ML is much subtler, though, and its analysis requires one to use carefully tailored store annotations in the corresponding game semantics [23].

Definition 4. *The fragment* $\mathrm{RML}_{\mathrm{EBVASS}}$ *consists of finitary* RML *terms-in-context of the form,* $x_1 : \theta_3, \ldots, x_n : \theta_3 \vdash M : \theta_0 \to \theta_2$, *where*

$$\theta_0 ::= \mathsf{unit} \mid \mathsf{int} \qquad \theta_1 ::= \theta_0 \mid \theta_0 \to \theta_1 \mid \mathsf{int\ ref}$$
$$\theta_2 ::= \theta_0 \mid \theta_1 \to \theta_0 \mid \mathsf{int\ ref} \qquad \theta_3 ::= \theta_0 \mid \theta_2 \to \theta_3 \mid \mathsf{int\ ref}$$

Example 5. The following term $\vdash M : \mathsf{int} \to (\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$ is in $\mathrm{RML}_{\mathrm{EBVASS}}$.

```
λx^int. let m = ref(0)
        in λf^(int→int). assert(even(!m)); if even(x) then m := 1;
                         let y=f(x) in m := x; y
```

We write $\mathbf{assert}(M)$ for $\mathbf{if}\, M\, \mathbf{then}\, ()\, \mathbf{else}\, \Omega$, where Ω is the divergent term $\mathbf{while}\, 1\, \mathbf{do}\, ()$. When applied to an integer x, the term yields a function of type $(\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$, which will apply its argument (a function $f : \mathsf{int} \to \mathsf{int}$) to x. However, owing to the assertion and the side effects, the behaviour of $M\,x$ is quite different from $\lambda f^{\mathsf{int}\to\mathsf{int}}.f\,x$. If x is even then only sequential (non-overlapping) uses of the function will be allowed. Thus, $\mathbf{let}\, g = M\, 0\, \mathbf{in}\, (\mathbf{let}\, a = g(\lambda x^{\mathsf{int}}.0)\, \mathbf{in}\, g(\lambda y^{\mathsf{int}}.0))$ terminates, whereas $\mathbf{let}\, g = M\, 0\, \mathbf{in}\, g(\lambda x^{\mathsf{int}}.g(\lambda y^{\mathsf{int}}.0))$ diverges. In contrast, when x is odd, $M\,x$ can only be called in a nested way and new calls become forbidden as soon as the first call returns. Thus, a typical usage pattern consists of a series of nested calls (of arbitrary depth) followed by the same number of returns. Consequently, $\mathbf{let}\, g = M\, 1\, \mathbf{in}\, g(\lambda x^{\mathsf{int}}.g(\lambda y^{\mathsf{int}}.0))$ terminates, whereas $\mathbf{let}\, g = M\, 1\, \mathbf{in}\, (\mathbf{let}\, a = g(\lambda x^{\mathsf{int}}.0)\, \mathbf{in}\, g(\lambda y^{\mathsf{int}}.0))$ diverges.

3 Game Semantics of RML

We use a presentation of call-by-value game semantics in the style of Honda and Yoshida [13], as opposed to Abramsky and McCusker's isomorphic model [3], as Honda and Yoshida's more concrete constructions lend themselves more easily to recognition by automata. We recall the following presentation of the game semantics for RML from [15].

An *arena* A is a triple $(M_A, \vdash_A, \lambda_A)$ where M_A is a set of *moves* where $I_A \subseteq M_A$ consists of *initial* moves, $\vdash_A \subseteq M_A \times (M_A \backslash I_A)$ is called the *enabling relation*, and $\lambda_A : M_A \to \{O, P\} \times \{Q, A\}$ a labelling function such that for all $i_A \in I_A$ we have $\lambda_A(i_A) = (P, A)$, and if $m \vdash_A m'$ then $(\pi_1 \circ \lambda_A)(m) \neq (\pi_1 \circ \lambda_A)(m')$ and $(\pi_2 \circ \lambda_A)(m') = A \Rightarrow (\pi_2 \circ \lambda_A)(m) = Q$. The function λ_A labels moves as

belonging to either *Opponent* or *Proponent* and as being either a *Question* or an *Answer*. Note that answers are always enabled by questions, but questions can be enabled by either a question or an answer. We will use arenas to model types. However, the actual games will be played over *prearenas*, which are defined in the same way except that initial moves are O-questions.

Three basic arenas are 0 (the empty arena), 1 (the arena containing a single initial move $\bullet$), and $\mathbb{Z}$ (has integers as moves, all of which are initial P-answers). In all cases, the enabling relation is empty. The constructions on arenas are defined in Figs. 4 and 5, where the lines represent enabling. Here we use $\overline{I_A}$ as an abbreviation for $M_A \backslash I_A$, and $\overline{\lambda_A}$ for the O/P-complement of λ_A. Intuitively $A \otimes B$ is the union of the arenas A and B, but with the initial moves combined pairwise. $A \Rightarrow B$ is slightly more complex. First we add a new initial move, $\bullet$. We take the O/P-complement of A, change the initial moves into questions, and set them to now be justified by $\bullet$. Finally, we take B and set its initial moves to be justified by A's initial moves. The final construction, $A \to B$, takes two arenas A and B and produces a prearena, as shown below. This is essentially the same as $A \Rightarrow B$ without the initial move $\bullet$.

$$
\begin{array}{ll}
M_{A \Rightarrow B} = \{\bullet\} \uplus M_A \uplus M_B & M_{A \otimes B} = I_A \times I_B \uplus \overline{I_A} \uplus \overline{I_B} \\
I_{A \Rightarrow B} = \{\bullet\} & I_{A \otimes B} = I_A \times I_B \\
\lambda_{A \Rightarrow B} = m \mapsto \begin{cases} PA & \text{if } m = \bullet \\ OQ & \text{if } m \in I_A \\ \overline{\lambda_A}(m) & \text{if } m \in \overline{I_A} \\ \lambda_B(m) & \text{if } m \in M_B \end{cases} & \lambda_{A \otimes B} = m \mapsto \begin{cases} PA & \text{if } m \in I_A \times I_B \\ \lambda_A(m) & \text{if } m \in \overline{I_A} \\ \lambda_B(m) & \text{if } m \in \overline{I_B} \end{cases} \\
\vdash_{A \Rightarrow B} = \{(\bullet, i_A) | i_A \in I_A\} & \vdash_{A \otimes B} = \{((i_A, i_B), m) | i_A \in I_A \wedge i_B \in I_B \\
\quad \cup \{(i_A, i_B) | i_A \in I_A, i_B \in I_B\} & \qquad \wedge (i_A \vdash_A m \vee i_B \vdash_B m)\} \\
\quad \cup \vdash_A \cup \vdash_B & \quad \cup (\vdash_A \cap (\overline{I_A} \times \overline{I_A})) \\
& \quad \cup (\vdash_B \cap (\overline{I_B} \times \overline{I_B}))
\end{array}
$$

$$
M_{A \to B} = M_A \uplus M_B \quad \lambda_{A \to B}(m) = \begin{cases} OQ & \text{if } m \in I_A \\ \overline{\lambda_A}(m) & \text{if } m \in \overline{I_A} \\ \lambda_B(m) & \text{if } m \in M_B \end{cases}
$$

$$
I_{A \to B} = I_A \qquad \vdash_{A \to B} = \{(i_A, i_B) | i_A \in I_A, i_B \in I_B\} \cup \vdash_A \cup \vdash_B
$$

Fig. 4. Arena and prearena constructions: definitions

We intend arenas to represent types, in particular $[\![\mathsf{unit}]\!] = 1$, $[\![\mathsf{int}]\!] = \mathbb{Z}$ (or a finite subset of $\mathbb{Z}$ for $\mathrm{RML_f}$), $[\![\mathsf{int\ ref}]\!] = [\![\mathsf{unit} \to \mathsf{int}]\!] \otimes [\![\mathsf{int} \to \mathsf{unit}]\!]$ and $[\![\theta_1 \to \theta_2]\!] = [\![\theta_1]\!] \Rightarrow [\![\theta_2]\!]$. A term-in-context $x_1 : \theta_1, \ldots, x_n : \theta_n \vdash M : \theta$ will be represented by a *strategy* for the prearena $[\![\theta_1]\!] \otimes \ldots \otimes [\![\theta_n]\!] \to [\![\theta]\!]$.

A *justified sequence* in a prearena A is a sequence of moves from A in which the first move is initial and all other moves m are equipped with a pointer to an earlier move m', such that $m' \vdash_A m$. A *play* s is a justified sequence which additionally satisfies the standard conditions of Alternation, Well-Bracketing,

and Visibility [3]. A *strategy* σ for prearena A is a non-empty, even-prefix-closed set of plays from A, satisfying the determinism condition: if $s\,m_1, s\,m_2 \in \sigma$ then $s\,m_1 = s\,m_2$. We can think of a strategy as being a playbook telling P how to respond by mapping odd-length plays to moves. A play is *complete* if all questions have been answered. Note that (unlike in the call-by-name case) a complete play is not necessarily maximal. We denote the set of complete plays in strategy σ by $\mathbf{comp}(\sigma)$.

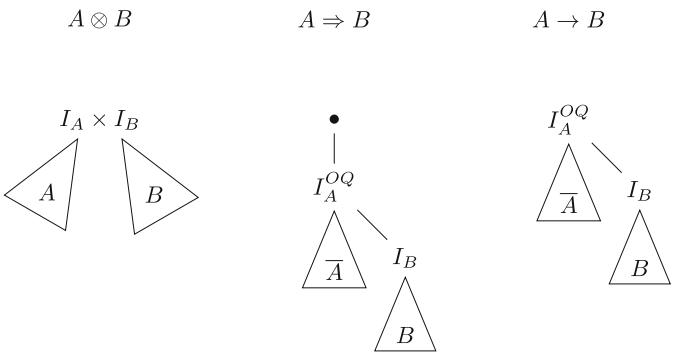


Fig. 5. Arena and prearena constructions, pictorially

In the game model of RML, a term-in-context $x_1 : \theta_1, \ldots, x_n : \theta_n \vdash M : \theta$ is interpreted by a strategy of the prearena $[\![\theta_1]\!] \otimes \ldots \otimes [\![\theta_n]\!] \to [\![\theta]\!]$. These strategies are defined by recursion over the syntax of the term. Free identifiers $x : \theta \vdash x : \theta$ are interpreted as *copy-cat* strategies where P always copies O's move into the other copy of $[\![\theta]\!]$, $\lambda x.M$ allows multiple copies of $[\![M]\!]$ to be run, application MN requires a form of parallel composition plus hiding and the other constructions can be interpreted using special strategies. The game-semantic model is fully abstract in the following sense.

Theorem 6 (Abramsky and McCusker [2,3]). *If* $\Gamma \vdash M : \theta$ *and* $\Gamma \vdash N : \theta$ *are* RML *terms then* $\Gamma \vdash M \cong N$ *iff* $\mathbf{comp}([\![M]\!]) = \mathbf{comp}([\![N]\!])$.

To represent the game semantics for the fragment $\mathrm{RML}_{\mathrm{EBVASS}}$, we need an automaton over an infinite alphabet which is equipped with a visibly pushdown stack. The shape of the prearenas for terms-in-context in this fragment is shown in Fig. 6.

Remark 7. We describe the intuitive meaning of various moves from the Figure. q_0 starts the evaluation of the term. a_0 stands for successful evaluation. q_1 invokes the resultant function with a base-type argument, while a_1 means that a value of type $(\beta \to \cdots \to \beta) \to \beta$ was generated. q_2 then corresponds to calling the value on a function argument, q_*^1, a_*^1, .., q_*^{m-1}, a_*^{m-1} represent interaction with that argument, while a_2 means that the call has returned.

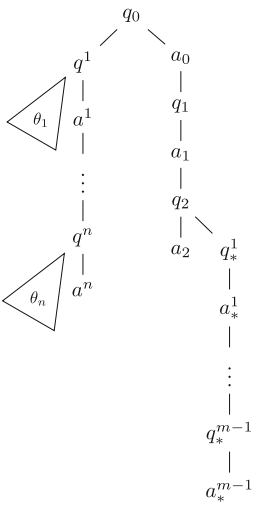
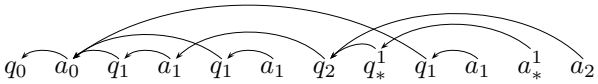


Fig. 6. Shape of prearena for $\theta_1 \to \cdots \to \theta_n \to \beta \vdash \beta \to (\beta_1 \to \cdots \to \beta_m) \to \beta$

Next we analyse the shape of non-empty complete plays in such arenas. At the beginning, each such play will contain a segment $q_0\, s\, a_0$ where s contains moves originating from the left-hand side of the arena. The unique occurrence of a_0 can be used to justify subsequent occurrences of q_1, each of which will have to be answered with a_1. Note that, due to the visibility condition, the moves between q_1 and the corresponding a_1 can only come from the left-hand side of the arena. It will be useful to think of each $q_1 \overset{\frown}{\;} a_1$-pair as defining a thread of play (moves made between q_1 and a_1 can then be said to occur in that thread).

Further, each a_1 can be used to justify subsequent occurrences of q_2, which we may think of as starting a subthread of the corresponding thread $q_1 \overset{\frown}{\;} a_1$. Note that in this case the justification pointer from q_2 is crucial in linking the q_2-subthread to the corresponding thread determined by $q_1 \overset{\frown}{\;} a_1$. We give a sample play below, which represents the interaction of the term $\lambda x^{\mathsf{unit}}.\lambda f^{\mathsf{unit}\to\mathsf{unit}}.fx$ with context $\mathbf{let}\, g = [\,]\, \mathbf{in}\, \mathbf{let}\, f_1 = g()\, \mathbf{in}\, \mathbf{let}\, f_2 = g()\, \mathbf{in}\, f_1(\lambda x^{\mathsf{unit}}.\mathbf{let}\, f_3 = g()\, \mathbf{in}\, ())$.



Observe that due to the well-bracketing condition and the availability of q_*^i moves, each thread can have a pushdown character. Thus, a play becomes an interleaving of pushdown threads subject to the global well-bracketing condition. This interleaving may switch between threads after any a_1, a_2, or q_*^i-move. Where a q_*^i-move is made, the corresponding q_2-subthread can only be returned to subject to the stack discipline. Furthermore, whenever O has the opportunity to start a new thread – after an a_1, a_2, or q_*^i-move, it can also create a new q_2-subthread by pointing at a visible occurrence of a_1. Later on we shall introduce an automata-theoretic model over infinite alphabets, called VPCMA, to capture such scenarios. The preceding play will correspond to the following data word

$$(q_0,n_0)(a_0,n_0)(q_1,n_1)(a_1,n_1)(q_1,n_2)(a_1,n_2)(q_2,n_1)(q_*^1,n_1)(q_1,n_3)(a_1,n_3)(a_*^1,n_1)(a_2,n_1)$$

where n_1, n_2, n_3 are elements of the infinite alphabet playing the rôle of thread identifiers (technically, they represent pointers from q_2).

There is one more complication due to the visibility condition. Note that once a_2 is played, it will remove the third q_1a_1 segment from the O-view and will effectively prevent the thread from generating future q_2-subthreads. Thus, the visibility condition restricts the way in which threads can be revisited to be compatible with the stack discipline. This constraint will motivate a variant of VPCMA, called *scoping VPCMA*.

4 Visibly Pushdown Class Memory Automata

In this section we introduce *visibly pushdown class memory automata* (VPCMA), which will be a convenient mechanism for capturing the game-semantic scenarios discussed at the end of the previous section.

VPCMA are a formalism over *data words*, i.e., elements of $(\Sigma \times \mathcal{D})^*$ where Σ is a finite alphabet of *data tags* and $\mathcal{D}$ is an infinite set of *data values*. VPCMA combine ideas from class memory automata (CMA) [7] and visibly pushdown automata (VPA) [4]. As with CMA, our VPCMA will have a class memory function that, for each data value seen in the run, will remember the state in which the data value was last seen. Following VPA, the input alphabet Σ will be partitioned into $\Sigma_{\mathtt{push}}$, $\Sigma_{\mathtt{pop}}$, and $\Sigma_{\mathtt{noop}}$, which determine the kind of stack action that is performed once letters from $\Sigma \times \mathcal{D}$ are being read. Stack actions will use elements of $\Gamma \times \mathcal{D}$, where Γ is a finite stack alphabet. The only subtlety in how these two kinds of automata are combined is in the contents of the stack: whenever an element of $\mathcal{D}$ will be involved in a push or pop, we shall require that it be equal to the element of $\mathcal{D}$ that is currently read by the machine. Thus, matching push- and pop-moves will always read the same data value. The data values on the stack can only be used in enforcing that the same data value that pushed an element to the stack is used to pop it off the stack.

Definition 8 (VPCMA). *Let $\Sigma = \Sigma_{\mathtt{push}} + \Sigma_{\mathtt{pop}} + \Sigma_{\mathtt{noop}}$ be finite and $Q_\perp = Q + \{\perp\}$. Fix an infinite dataset $\mathcal{D}$. A* visibly pushdown class memory automaton *is a tuple $\langle Q, \Sigma, \Gamma, \Delta, q_0, F_G, F_L \rangle$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, $F_G \subseteq F_L \subseteq Q$ are sets of* globally *and* locally *accepting states respectively, Γ a finite stack alphabet and Δ is the transition relation, where:*

$$\Delta \subseteq \big(Q \times Q_\perp \times (\Sigma_{\mathtt{push}} \cup \Sigma_{\mathtt{pop}}) \times \Gamma \times Q\big) \cup \big(Q \times Q_\perp \times \Sigma_{\mathtt{noop}} \times Q\big).$$

We explain the workings of a VPCMA below. A configuration is a triple (q, f, S) where $q \in Q$ is the current state, $f : \mathcal{D} \to Q_\perp$ is a *class memory function*, and $S \in (\mathcal{D} \times \Gamma)^*$ is the stack. The initial configuration is (q_0, f_0, ϵ) where f_0 maps all data values to $\perp$. A configuration (q, f, S) is *accepting* if $q \in F_G$, $f(d) \in F_L \cup \{\perp\}$ for all $d \in \mathcal{D}$, and $S = \epsilon$. On reading an input letter $(a, d) \in \Sigma \times \mathcal{D}$ whilst in configuration (q, f, S) the automaton can follow transitions as follows:

- if $a \in \Sigma_{\texttt{push}}$ the automaton can follow a transition $(q, f(d), a, \gamma, q')$ to configuration $(q', f[d \mapsto q'], S \cdot (d, \gamma))$.
- if $a \in \Sigma_{\texttt{pop}}$ and $S = S' \cdot (d, \gamma)$ the automaton can follow a transition $(q, f(d), a, \gamma, q')$ to configuration $(q', f[d \mapsto q'], S')$.
- if $a \in \Sigma_{\texttt{noop}}$ the automaton can follow a transition $(q, f(d), a, q')$ to configuration $(q', f[d \mapsto q'], S)$.

Acceptance of words is then defined in the normal way, with a word being accepted just if there is a run of the word from the initial configuration to an accepting configuration. Determinism is also defined in the normal way. That is, a VPCMA is deterministic just if the following conditions all hold:

(i) $(q, s, a_{\texttt{push}}, \gamma, p), (q, s, a_{\texttt{push}}, \gamma', p') \in \Delta \;\Rightarrow\; \gamma = \gamma', p = p'$;
(ii) $(q, s, a_{\texttt{pop}}, \gamma, p), (q, s, a_{\texttt{pop}}, \gamma, p') \in \Delta \;\Rightarrow\; p = p'$; and
(iii) $(q, s, a_{\texttt{noop}}, p), (q, s, a_{\texttt{noop}}, p') \in \Delta \;\Rightarrow\; p = p'$.

In our translation from RML, we shall rely on *weak VPCMA*, in which all states are locally accepting, i.e. $F_L = Q$. Then a configuration is final if a global accepting state has been reached and the stack is empty. Although for class memory automata (CMA), there is a significant gap between the complexity of normal CMA and weak CMA emptiness (corresponding essentially to the difference between reachability and coverability in vector addition systems) [10], there is no similar gap for VPCMA. The emptiness problem for VPCMA can be easily reduced to that for weak VPCMA by constructing, for a given VPCMA, a weak VPCMA which will at the very beginning guess all the data values to be used in an accepting run, push them on the stack one by one and, at the very end, verify the local acceptance conditions for each data value during pops.

Proposition 9. *Emptiness of VPCMA can be reduced to emptiness of weak VPCMA.*

Using standard product constructions, in the same way as for weak CMA [10], one can show that weak VPCMA are closed under union and intersection. Deterministic weak VPCMA are also closed under complementation (by reversing accepting states) but the complement needs to be taken with respect to the set of "well-bracketed" words generated by the grammar

$$W ::= \quad \epsilon \;\mid\; (a_{\texttt{noop}}, d) \cdot W \;\mid\; (a_{\texttt{push}}, d) \cdot W \cdot (a_{\texttt{pop}}, d) \cdot W$$

where d ranges over $\mathcal{D}$, and $a_{\texttt{push}}$, $a_{\texttt{pop}}$, and $a_{\texttt{noop}}$ range over $\Sigma_{\texttt{push}}$, $\Sigma_{\texttt{pop}}$, and $\Sigma_{\texttt{noop}}$ respectively. The closure properties make it possible to reduce deterministic VPCMA inclusion and equivalence to VPCMA emptiness.

We wrap this section up with the introduction of a special kind of VPCMA, called *scoping* VPCMA (SVPCMA). This variant is meant to reflect the shape of plays analysed at the end of Sect. 3 particularly well. Its definition is identical to that of VPCMA. The difference is in how the runs are defined, and as a result in the languages recognised. For SVPCMA, a configuration keeps track not just of the current state, class memory function, and stack, but also of a set

of "visible" data values. The idea is that when a data value is first read after a push-move but before that move's corresponding pop-move, this data value will only be usable until that pop-move – preventing the data value from "leaking" into other parts of the run. Consequently, a tree hierarchy is imposed on the use of data values. Although this may seem a substantial restriction at first, scoping VPCMA turn out to have identical algorithmic properties to normal VPCMA.

Definition 10. *A* scoping *VPCMA (SVPCMA) is a tuple* $\langle Q, \Sigma, \Gamma, \Delta, q_0, F_G, F_L \rangle$ *of the same construction as a VPCMA.*

In contrast to VPCMA configuration, an SVPCMA configuration is a tuple (q, f, V, S) where q and f are states and class memory functions as before, $V \subset_{\text{fin}} \mathcal{D}$ is the set of visible data values, and $S \in (\mathcal{D} \times \Gamma \times \mathcal{P}_{\text{fin}}(\mathcal{D}))^*$. The initial configuration is $(q_0, f_0, \emptyset, \epsilon)$, and a configuration is accepting just in the conditions set for normal VPCMA (i.e. no restrictions on V). On reading an input letter (a, d) whilst in configuration (q, f, V, S), if $f(d) = \perp$ or $d \in V$ the automaton can follow transitions as follows:

- if $a \in \Sigma_{\texttt{push}}$ the automaton can follow a transition $(q, f(d), a, \gamma, q')$ to configuration $(q', f[d \mapsto q'], V \cup \{d\}, S \cdot (d, \gamma, V \cup \{d\}))$.
- if $a \in \Sigma_{\texttt{pop}}$ and $S = S' \cdot (d, \gamma, V')$ the automaton can follow a transition $(q, f(d), a, \gamma, q')$ to configuration $(q', f[d \mapsto q'], V', S')$.
- if $a \in \Sigma_{\texttt{noop}}$ the automaton can follow a transition $(q, f(d), a, q')$ to configuration $(q', f[d \mapsto q'], V \cup \{d\}, S)$.

Note that if $f(d) \neq \perp$ and $d \notin V$, the automaton cannot transition!

Weakness and determinism for SVPCMA are defined in the usual way. And we can obtain the same result collapsing weakness as for normal VPCMA:

Proposition 11. *Emptiness of SVPCMA can be reduced to emptiness of weak SVPCMA.*

Proof (Sketch). The idea for this construction is similar to that for VPCMA, but this time we cannot just read all of the data values at the start of the run, and check them at the end. Instead, whenever a new data value would be introduced we first introduce it with a push-move; and when that value is popped, we check that it is in a locally accepting state, and prevent it from being used again.

Similarly, all of the closure constructions that work for VPCMA also work for SVPCMA (though this time closure is with respect to well-bracketed words that are consistent with the SVPCMA restriction). In any case, the equivalence problem for deterministic SVPCMA can also be reduced to SVPCMA emptiness.

Next we discuss why the emptiness problems for VPCMA and SVPCMA are interreducible. Owing to the defining restriction for SVPCMA, not all languages recognisable by VPCMA are recognisable by SVPCMA, and vice versa. Hence, there cannot be effective translations between VPCMA and SVPCMA that preserve recognisability. However, we have

Proposition 12. *VPCMA and SVPCMA emptiness problems are interreducible.*

Proof. To reduce emptiness of VPCMA to that of SVPCMA we employ a similar trick to that used to reduce VPCMA to weak VPCMA: we begin by having the automaton read all of the data values that are going to be used in the run, then running the automaton as normal, with calls for fresh data values replaced with calls for data values seen at the start of the run.

To reduce emptiness of SVPCMA to that of VPCMA we employ a similar trick to that used to reduce SVPCMA to weak SVPCMA: whenever a data value is first read we insert a dummy push-move, which must be popped before any containing push-move is popped. When the dummy push-move is popped, we prevent that data value from being read again.

In Sect. 7 we show that VPCMA (SVPCMA) emptiness is recursively equivalent to reachability in extended branching VASS [17]. In the next section, we use SVPCMA to represent the game semantics of $\mathrm{RML}_{\mathrm{EBVASS}}$.

5 $\mathrm{RML}_{\mathrm{EBVASS}}$ to VPCMA

In this section we prove

Theorem 13. *Observational equivalence of* $\mathrm{RML}_{\mathrm{EBVASS}}$*-terms is reducible to the emptiness problem for VPCMA.*

The result, in conjunction with results of Sect. 7 will imply the left-to-right implication in Theorem 1. To establish Theorem 13, we rely on the following crucial lemma.

Lemma 14. *For any* $\mathrm{RML}_{\mathrm{EBVASS}}$*-term* $\Gamma \vdash M$*, there exists a weak deterministic SVPCMA* $\mathcal{A}^M$ *whose language is a faithful representation of* $\mathbf{comp}([\![\Gamma \vdash M]\!])$*.*

As discussed in Sect. 4, SVPCMA equivalence can be reduced to VPCMA emptiness, so the Lemma implies Theorem 13. We shall prove the Lemma by induction for terms in canonical form.

Definition 15. *An* RML *term is in* canonical form *if it is generated by the following grammar:*

$$\begin{aligned}\mathbb{C} ::= {} & () \mid i \mid x^{\beta} \mid \mathbf{succ}(x^{\beta}) \mid \mathbf{pred}(x^{\beta}) \mid \mathbf{if}\, x^{\beta}\, \mathbf{then}\, \mathbb{C}\, \mathbf{else}\, \mathbb{C} \mid \\ & x^{\mathsf{int\ ref}} := y^{\mathsf{int}} \mid\ !x^{\mathsf{int\ ref}} \mid \mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}\, \mathbb{C} \mid \mathbf{mkvar}(\lambda u^{\mathsf{unit}}.\mathbb{C}, \lambda v^{\mathsf{int}}.\mathbb{C}) \mid \\ & \mathbf{while}\, \mathbb{C}\, \mathbf{do}\, \mathbb{C} \mid \lambda x^{\theta}.\mathbb{C} \mid \mathbf{let}\, x^{\beta} = \mathbb{C}\, \mathbf{in}\, \mathbb{C} \mid \mathbf{let}\, x = z\, y^{\beta}\, \mathbf{in}\, \mathbb{C} \mid \\ & \mathbf{let}\, x = z\, (\lambda x^{\theta}.\mathbb{C})\, \mathbf{in}\, \mathbb{C} \mid \mathbf{let}\, x = z\, \mathbf{mkvar}(\lambda u^{\mathsf{unit}}.\mathbb{C}, \lambda v^{\mathsf{int}}.\mathbb{C})\, \mathbf{in}\, \mathbb{C}\end{aligned}$$

It can be shown [14] that, for any RML term $\Gamma \vdash M : \theta$ there is a term $\Gamma \vdash N : \theta$ in canonical form, effectively constructible from M, such that $[\![\Gamma \vdash M]\!] = [\![\Gamma \vdash N]\!]$ (for the most part, the conversions involve let-commutations and β-reduction).

Next we explain how justification pointers from games will be handled. Pointers from answers need not be represented explicitly, because they can be

reconstructed uniquely from the underlying sequences of moves via the Well-Bracketing condition. Pointers from questions may need to be represented, but sometimes they too are uniquely recoverable thanks to the Visibility condition, when at most one justifier is guaranteed to occur in the relevant view. For O-questions, this was always the case in the fragment considered in [15], called the *O-strict fragment*. $\mathrm{RML}_{\mathrm{EBVASS}}$ is an extension of that fragment and some O-pointers will need to be represented explicitly, but fortunately these are only pointers from moves marked q_2 in Fig. 6. As already hinted at the end of Sect. 3, we shall use data values to handle the issue as follows.

- The unique q_0-move, and each q_1-move will take a fresh data value.
- All a_i-moves will take the same data value as their justifying q_i-move.
- Each q_2-move, and all hereditarily justified moves, will take the same data value as their justifying a_1-move.
- Each move corresponding to the types of free variables in the term will take the same data value as the preceding move.

Because q_2-moves are labelled with the same data value as their justifiers, the problematic O-pointers are clearly represented by the above scheme. As concerns P-pointers, there are also cases in which pointers from P-moves have to be represented explicitly, because there may be two potential justifiers in the relevant P-view. Fortunately, the problems are of the same kind as those for the O-strict fragment and can be handled using the marking technique used in [15,18].

Next we discuss the automata constructions, focussing on the new cases with respect to [15], i.e. when the term is of type $\beta \to (\beta \to \cdots \to \beta) \to \beta$. In other cases, i.e. $()$, i, x^β, $\mathbf{succ}(x^\beta)$, $\mathbf{pred}(x^\beta)$, $\mathbf{if}\, x^\beta\, \mathbf{then}\, \mathbb{C}\, \mathbf{else}\, \mathbb{C}$, $x^{\mathsf{int\ ref}} := y^{\mathsf{int}}$, $!x^{\mathsf{int\ ref}}$, and $\mathbf{mkvar}(\lambda x^{\mathsf{unit}}.\mathbb{C}, \lambda y^{\mathsf{int}}.\mathbb{C})$, $\mathbf{while}\, \mathbb{C}\, \mathbf{do}\, \mathbb{C}$ we can rely on the constructions from [15], as they produce visibly pushdown automata, which can be easily be upgraded to SVPCMA by annotating each move with a dummy data value. $\lambda x.M$ The most important case is that of λ-abstraction. In the case where $\lambda x.M$ is not of type $\beta \to (\beta \to \cdots \to \beta) \to \beta$, this has already been covered by the VPA constructions. We therefore can assume this is the final lambda abstraction in the term, and so x is of type $\beta \in \{\mathsf{int}, \mathsf{unit}\}$ and M's type is of the shape $(\beta \to \cdots \to \beta) \to \beta$.

Then the key idea of this construction is that the strategy for $\lambda x.M$, after the unique a_0-move, is an interleaving of multiple strategies for M. Since we can handle M with a VPA [15], each q_1-move corresponds to starting a new VPA running. SVPCMA allow us to simulate multiple VPAs, each identified by its own data value. The well-bracketing constraint on plays is enforced by the single stack discipline of the SVPCMA, while the visibility condition on O-pointers is checked by the scoping restriction on SVPCMA.

Before we give the formal definition of the SVPCMA for $\lambda x.M$, we analyse the plays in $[\![\lambda x.M]\!]$ in more detail. O starts by playing an initial move γ, to which P plays the unique response a_0. O then starts a q_1-thread with a move i_x corresponding to the value of x. Play then in that thread continues as in $[\![M]\!]$ with initial move (γ, i_x). However, at any point after P has played an a_1, q_*^j,

or a_2 move, O may switch to another thread (new or existing), subject to that thread (i.e. the $q_1 \frown a_1$ moves of that thread) being visible.

For the construction, we know that there is a family of VPA, $(\mathcal{A}_i^M)$, where $\mathcal{A}_i^M$ recognises the complete plays from $[\![M]\!]$ that start with initial move i (the move i is omitted). We note that these initial moves have an x-component, as x is a free variable of ground type in M, hence we can think of the initial moves as having the form (γ, i_x), where i_x is the part that corresponds to x. We make a further assumption on the $(\mathcal{A}_i^M)$, that the states reachable by following a transition with a Σ-label corresponding to a a_1, q_*^j, or a_2 move can only be reached by following transitions with those Σ-labels. We write N_i for these states. (Note that it is straightforward to convert a VPA without this property to one with it.) Further we note that these states, due to the plays possible, will only have no-op and pop transitions from them.

Hence we construct the automata $(\mathcal{A}_\gamma^{\lambda x.M})$ as follows.

- The set of states of $\mathcal{A}_\gamma^{\lambda x.M}$ is formed of two new states, (1) and (2) together with the disjoint union of the states from each $\mathcal{A}^M_{(\gamma,i_x)}$ (for each possible value i_x).
- The initial state is the new state (1).
- The set of globally accepting states is the union of the sets of accepting states from each $\mathcal{A}^M_{(\gamma,i_x)}$ together with the new state (2).
- The transitions are defined as follows:
  - There is a (no-op) transition (1) $\xrightarrow{a_0,\bot}$ (2)
  - For each i_x there is a (no-op) transition (2) $\xrightarrow{i_x,\bot} q_{i_x}$ where q_{i_x} is the initial state of $\mathcal{A}^M_{(\gamma,i_x)}$
  - For each $\mathcal{A}^M_{(\gamma,i_x)}$:
    - For each no-op transition $q_1 \xrightarrow{m} q_2$ inside $\mathcal{A}^M_{(\gamma,i_x)}$, there is a (no-op) transition $q_1 \xrightarrow{m,q_1} q_2$
    - For each push/pop transition $q_1 \xrightarrow{m,\sigma} q_2$ inside $\mathcal{A}^M_{(\gamma,i_x)}$, there is a (push/pop resp.) transition $q_1 \xrightarrow{m,q_1,\sigma} q_2$
  - For each state q_1, q_2 in $\bigcup_{i_x} N_{(\gamma,i_x)}$ and each no-op transition $q_2 \xrightarrow{m} q_3$ in the constituent automaton there is a transition $q_1 \xrightarrow{m,q_2} q_3$. Similarly for each pop transition $q_2 \xrightarrow{a,\sigma} q_3$ we have the transition $q_1 \xrightarrow{a,q_2,\sigma} q_3$. This allows for changing between threads at the appropriate points.

The remaining cases concern $\mathbf{let}\, x = \ldots \,\mathbf{in}\, M$ and adaptations of the corresponding cases in the O-strict constructions in the O-strict case [14,15]. Crucially, whilst these constructions all allow the "interruption" of $[\![M]\!]$ to make plays corresponding to x, the strategy for x can be recognised by a normal VPA and so the interruptions do not disturb the data value being used. Hence the adaptations from the O-strict case are straightforward. We discuss two of the cases in more detail.

$\mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}\, M$ The states of $\mathcal{A}_\gamma^{\mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}\, M}$ is equal to the states of $\mathcal{A}_\gamma^M$ crossed with the finitary fragment of $\mathbb{N}$ being used. We refer to the new finitary

fragment as the x-component of the state. The new initial state is the old initial state with x-component 0. Transitions are generally preserved, without altering the x-component, except $write_x(i)$-transitions now change the x-component to i, and answers to $read_x$-transitions must match the current x-component (other answer transitions are removed). For every (maximal) sequence of x-transitions out of a state, we now replace that sequence with a silent transition, which we then eliminate (and alter the required signature of the data value accordingly). Since the data value being read cannot change in sequences of x-transitions, this is a straightforward operation.

$\mathbf{let}\, x^\beta = N \,\mathbf{in}\, M$ $[\![\mathbf{let}\, x^\beta = N \,\mathbf{in}\, M]\!]$ first evaluates N, i.e. runs as $[\![N]\!]$ until a value is returned for x, then begins running as $[\![M]\!]$ in which that value of x was provided in the first move.

Since N is of type β, there are VPA $(\mathcal{A}^N_\gamma)$ representing $[\![\Gamma \vdash N]\!]$. Further since x is free in M the initial moves in M have an x-component, so we have a family of SVPCMA $(\mathcal{A}^M_{(\gamma,i_x)})$. The automata construction for the term is then a fairly straightforward concatenation of the the automata for N and M, with which copy of $\mathcal{A}^M$ used being determined by the outcome of $\mathcal{A}^N$. The only difficulty is adding the data values to the automaton for N, but this is straightforward as only one data value is used for the entire run of N.

6 VPCMA to $\mathbf{RML_{EBVASS}}$

So far we have shown that observational equivalence of terms in $\mathrm{RML_{EBVASS}}$ is reducible to emptiness of SVPCMA. In this section we show that the converse is also true.

To reduce SVPCMA emptiness to observational equivalence of $\mathrm{RML_{EBVASS}}$-terms, we will first alter the given SVPCMA to make the reduction to RML-terms easier. We already saw, in Sect. 4, that given an SVPCMA it is possible to construct a weak SVPCMA with equivalent emptiness problem.

Now, given a weak SVPCMA $\mathcal{A}$, by doubling the states and stack alphabet, it is straightforward to construct another weak SVPCMA, $\mathcal{A}'$, recognising the same languages as $\mathcal{A}$ such that whether or not the stack is empty is stored in the state of the automaton. Hence, the emptiness of $\mathcal{A}'$ is determined just by whether or not a globally accepting state is reachable.

How then, do we construct the RML terms from $\mathcal{A}'$? We shall represent each data value by a single $q_1 \overset{\frown}{\ } a_1$-thread. Hence, a transition reading a new data value will be represented by O playing q_1 and P responding with a_1. The class memory function's value for this data value will then be stored in a local variable with suitable scope. Noop-moves not taking a fresh data value can then be made by playing $q_2 \overset{\frown}{\ } a_2$-moves justified by the $q_1 \overset{\frown}{\ } a_1$ corresponding to the data value. When the q_2-move is played, the term can update the class memory function as required.

Push-moves will be represented by $q_2 \overset{\frown}{\ } q_*$-moves, with the stack letter stored locally. If the push-move introduces a fresh data value, the $q_1 \overset{\frown}{\ } a_1$-thread must be created first, and then immediately followed by the $q_2 \overset{\frown}{\ } q_*$-moves. Pop-moves

will be represented by $a_* a_2$ pairs. Note that the Well-Bracketing condition will enforce the stack discipline during the simulation. Furthermore, as we saw in the previous section, the visibility condition of the plays will correspond precisely to the scoping condition of SVPCMA, that restricts use of data values first seen inside pushes.

In the term, we will need O to choose which transition is fired next. We will do this by alternating O's plays between those that correspond to transitions of the SVPCMA as described already, and a simple q_1-move that provides as int-input, which transition will be fired next. The term, using a global variable, can keep track of whether the next O-move should be providing input, or simulating a transition.

Using these ideas, we prove that the representation scheme can be implemented using $\mathrm{RML}_{\mathrm{EBVASS}}$-terms.

Proposition 16. *Given a weak SVPCMA such that the automaton can only arrive at a final state with an empty stack, there are* $\mathrm{RML}_{\mathrm{EBVASS}}$*-terms*

$$\vdash M, N : \mathsf{int} \to (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$$

such that the language recognised by the automaton is non-empty iff M *and* N *are not observationally equivalent.*

The only difference between M and N above is that one of them will diverge on reaching the final state whereas the other will carry on simulating the last step. In the above we have used an int-type, to make it easy for P to ask the environment (O) which transition should be fired next. We note that we could have used only unit-types, using a different scheme for O-choices. For example, at the very beginning we could introduce as many $q_1 \overset{\frown}{a_1}$ segments as there are transitions and O-choice could be represented by playing a $q_2 \overset{\frown}{a_2}$ justified by one of the a_1 (so O-choice would be represented by the choice of a justifer, one of the special a_1's). Thus, the result can also be shown to hold for $\mathsf{unit} \to (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$.

Thus we have shown that $\mathrm{RML}_{\mathrm{EBVASS}}$ observational equivalence and VPCMA emptiness are recursively equivalent.

7 VPCMA and EBVASS

In this section we show that VPCMA emptiness and EBVASS reachability are interreducible. We first review *extended branching VASS* (EBVASS), which were introduced in [17] to analyse a two-variable fragment of first-order logic over data trees, and shown to be equivalent to a form of data tree automaton.

EBVASS are slightly more powerful than *branching VASS* (BVASS) [11], whose reachability problem is not known to be decidable. Thus, we begin our review with BVASS, which extend VASS where, in addition to the standard transitions affecting the counter values and the state, there are "split" transitions, which split the current counter values into two copies of the current VASS, each copy then transitioning to a pre-given state. These copies must then complete their runs independently. Formally:

Definition 17 (BVASS). *A (top-down)* branching vector addition system with states (BVASS) *is a tuple* $(Q, q_0, L, k, \Delta_u, \Delta_s)$ *where* Q *is a finite set of states,* $q_0 \in Q$ *is the initial (root) state,* $L \subseteq Q$ *is the set of target (leaf) states,* $k \in \mathbb{N}$ *is the number of counters (dimension of the BVASS), and* Δ_u *and* Δ_s *are the unary and split transition relations respectively. The unary and split relations are of the forms:*

$$\Delta_u \subseteq (Q \times \mathbb{N}^k) \times (Q \times \mathbb{N}^k) \qquad \Delta_s \subseteq (Q \times \mathbb{N}^k) \times (Q \times \mathbb{N}^k) \times (Q \times \mathbb{N}^k)$$

We may write unary transitions, $(q_1, \bar{v_1}, q_2, \bar{v_2}) \in \Delta_u$*, and split transitions,* $(q_1, \bar{v_1}, q_2, \bar{v_2}, q_3, \bar{v_3}) \in \Delta_s$*, in the following ways:*

$$\textit{(unary)} \quad \frac{q_1, \bar{v} + \bar{v_1}}{q_2, \bar{v} + \bar{v_2}} \qquad \textit{(split)} \quad \frac{q_1, \bar{v} + \bar{v}' + \bar{v_1}}{q_2, \bar{v} + \bar{v_2} \qquad q_3, \bar{v}' + \bar{v_3}}$$

These representations reflect the runs of BVASS, which we now define. A configuration of a BVASS is a pair $(q, \bar{v})$ where $q \in Q$ and $\bar{v} \in \mathbb{N}^k$. A run of a BVASS is a (finite) tree labelled with configurations, such that each node has at most two children, with the following conditions:

- if a node labelled with $(q, \bar{v})$ has precisely one child node, then there is a transition $(q, \bar{u_1}, q', \bar{u_2}) \in \Delta_u$ such that $\bar{v} - \bar{u_1} \in \mathbb{N}^k$ and the child node is labelled with $(q', \bar{v} - \bar{u_1} + \bar{u_2})$.
- if a node labelled with $(q, \bar{v})$ has two child nodes, then there is a transition $(q, \bar{u_1}, q', \bar{u_2}, q'', \bar{u_3}) \in \Delta_s$ such that there exist $\bar{v_1}, \bar{v_2} \in \mathbb{N}^k$ such that $\bar{v} = \bar{v_1} + \bar{v_2} + \bar{u_1}$ and the left child node is labelled $(q', \bar{v_1} + \bar{u_2})$ and the right child node is labelled $(q'', \bar{v_2} + \bar{u_3})$.

A run is *accepting* just if every leaf node's label is $(q, \bar{0})$ for some $q \in L$. The reachability problem asks whether there is an accepting run of the BVASS with root configuration $(q_0, \bar{0})$.

We note that this is a strong form of BVASS, where several operations may be performed in one step: multiple increments and decrements. It is possible for unary transitions to be able to only make a single increment or decrement, and for split transitions to make no increments or decrements. It is clear that this more powerful presentation does not change the power of the model, but it allows us a slightly more concise reduction from VPCMA.

We now move to give a definition of EBVASS. These were introduced in [17], and extend BVASS with the ability to split counters in more complex ways when a split transition is made.

Definition 18 (EBVASS). *An* extended branching vector addition system with states (EBVASS) *is a tuple* $(Q, q_0, L, k, \Delta_u, \Delta_s, C)$ *where* $(Q, q_0, L, k, \Delta_u, \Delta_s)$ *is a BVASS and* $C \subseteq \{1, \ldots, k\}^3$ *is the set of* constraints.

Each constraint (i, j, k) can fire *any* number of times when a split transition is made, and for each time it fires it will decrement the ith counter (pre-splitting), and then increment the jth counter in the left-hand branch and the kth counter

in the right-hand branch. Formally, this means that runs are again finite labelled trees, with the rules for single-child nodes as for BVASS, but the following extended rule for nodes with two children.

– Suppose $C = \{c_1, \ldots, c_m\}$. If a node labelled $(q, \bar{v})$ has two child nodes then there is a transition $(q, \bar{u_1}, q', \bar{u_2}, q'', \bar{u_3}) \in \Delta_s$, $n_1, \ldots, n_m \in \mathbb{N}$, and $\bar{v_1}, \bar{v_2} \in \mathbb{N}^k$ such that $\bar{v} = \bar{v_1} + \bar{v_2} + \Sigma(n_i \cdot \bar{e}_{\pi_1(c_i)})$, and the left child node is labelled $(q', \bar{v_1} + \Sigma(n_i \cdot \bar{e}_{\pi_2(c_i)}))$, and the right child node is labelled $(q'', \bar{v_2} + \Sigma(n_i \cdot \bar{e}_{\pi_3(c_i)}))$, where the vector $\bar{e}_l \in \mathbb{Z}^k$ is 1 in position l and 0 elsewhere.

Again, a run is accepting just if each leaf node is labelled with a configuration $(q, \bar{0})$ where $q \in L$, and the *reachability problem* asks whether there is an accepting run with root node labelled $(q_0, \bar{0})$.

Remark 19. We work with a top-down version of EBVASS, as this formulation is more convenient for capturing the correspondence with VPCMA. In the language of [17, Sect. 5], our definition of runs corresponds to the non-commutative treatment of constraints.

7.1 From VPCMA to EBVASS

Theorem 20. *The emptiness problem for VPCMA is reducible to the reachability problem for EBVASS.*

We first give the central ideas behind the reduction.

– The states of the EBVASS will correspond to pairs of states of the VPCMA. If a position in the tree has a configuration with state (q, q') this will mean that the subtree under this position represents a stack-neutral run of the VPCMA from state q to q', i.e. all elements pushed on the stack will subsequently be removed.
– The counters in the EBVASS will correspond to pairs of states of the VPCMA. Each increment of a counter corresponding to the pair (q, q') in a position in a tree will (roughly) mean that there is a data value d with $f(d) = q$ that becomes a data value with $f(d) = q'$ within that subtree (and this needs to be borne out within the subtree).
– No-op moves in the VPCMA will be modelled by unary transitions in the EBVASS, adjusting the current state and counters appropriately.
– Push and pop moves will be modelled by split-transitions, with a single split-transition representing both the push and the pop move. The left-hand branch will correspond to the part of the run between the push and pop moves, whilst the right-hand branch corresponds to the moves after. Constraints allow data values to be split into what happens to them within the branch and what happens to them after.

We now give a formal account of the reduction. W.l.o.g. (Proposition 9) we work with weak VPCMA. Suppose $\mathcal{A} = \langle Q, \Sigma, \Gamma, \Delta, q_0, \{q_f\} \rangle$ is a weak

VPCMA[4]. We shall construct an EBVASS $\mathcal{E}_\mathcal{A}$ such that its reachability problem is a yes-instance iff $\mathcal{L}(\mathcal{A}) \neq \emptyset$ as follows.

We let the set of states of $\mathcal{E}_\mathcal{A}$ be $P = Q \times Q$, the initial state (q_0, q_f), the set of leaf states $L = \{(q, q) : q \in Q\}$. We set the number of counters $k = |Q_\perp \times Q|$, with a counter corresponding to each pair $(q, q') \in Q_\perp \times Q$. For each such pair, we use the notation $c_{q,q'}$ for the counter corresponding to that pair, and $\bar{e}_{q,q'}$ for the vector in $\mathbb{Z}^k$ with a 1 in position $c_{q,q'}$ and 0 elsewhere. The set of constraints contains $(c_{q,q''}, c_{q,q'}, c_{q',q''})$ for each $q, q', q'' \in Q_\perp$.

The transition relation for $\mathcal{E}_\mathcal{A}$ is given as follows:

- For each transition $(q, s, a, q') \in \Delta$, where $a \in \Sigma_{\texttt{noop}}$, we have:

$$(\text{NO-OP}) \quad \frac{(q,p), \bar{v} + \bar{e}_{s,s'}}{(q',p), \bar{v} + \bar{e}_{q',s'}}$$

- For each pair of transitions (q_1, s, a, γ, q_2) and $(q_3, s', b, \gamma, q_4)$ where $a \in \Sigma_{\texttt{push}}$ and $b \in \Sigma_{\texttt{pop}}$, we have:

$$(\text{PUSH-POP}) \quad \frac{(q_1,p), \bar{v_1} + \bar{v_2} + \bar{e}_{s,s''}}{(q_2,q_3), \bar{v_1} + \bar{e}_{q_2,s'} \qquad (q_4,p), \bar{v_2} + \bar{e}_{q_4,s''}}$$

  (Note that the above is a slight abuse of notation: split rules cannot also include increments[5]. However, it is straightforward to implement the above using unary transitions before and after the split, though to do this additional states must be introduced to keep track - we leave this out for clarity.)
- For every $x \in Q \times Q$ and $q \in Q$ we have the rule

$$(\text{DECREMENT}) \quad \frac{x, \bar{v} + \bar{e}_{q,q}}{x, \bar{v}}$$

  (This rule allows counters corresponding to data values which have "reached their required destination" to be decremented.)
- For every $x \in Q \times Q$ and $q \in Q$:

$$(\text{INCREMENT}) \quad \frac{x, \bar{v}}{x, \bar{v} + \bar{e}_{\perp,q}}$$

  (This rule makes it possible to add a new class along with its evolution profile, from $\perp$ to some state q.)

One can show that $\mathcal{L}(\mathcal{A}) \neq \emptyset$ iff there is a run of $\mathcal{E}_\mathcal{A}$ reaching the target configurations.

[4] We assume the set of globally accepting states to be a singleton merely for convenience - it is trivial to adjust.

[5] Actually there is another abuse of notation: the $\bar{v}_1$ and $\bar{v}_2$ may be altered by the constraints yet that is not mentioned in the rule.

7.2 From EBVASS to SVPCMA

Here we show that VPCMA emptiness is at least as hard as the reachability problem for EBVASS. W.l.o.g. (Proposition 12) we do this by reducing EBVASS reachability to SVPCMA emptiness. The key idea is that words over a pushdown alphabet can be viewed as trees by viewing them as their construction trees when generated by the grammar:

$$W ::= \epsilon \mid a_{\mathtt{noop}} \cdot W \mid a_{\mathtt{push}} \cdot W \cdot a_{\mathtt{pop}} \cdot W$$

Hence, a push-pop pair of moves correspond to a split-transition of the EBVASS, with the word occurring between the push and pop-moves corresponding to the left-hand branch, and the word after the pop-move corresponding to the right-hand branch. Our reduction argument will represent counter values as the number of data values with an appropriate class memory function value, and the EBVASS state can simply be stored as the SVPCMA state. The scoping visibility condition on runs will prevent increments made in the left-hand branch (from some split) being used in the right-hand branch. The only difficulty in the reduction is the handling of constraints: but for this we can use the stack again. After the push-move of a split-transition, we will be able to fire transitions corresponding to the constraints. Given a constraint (i, j, k) the corresponding push transition will take a data value where the class memory function remembers it as belonging to counter i, change it to belong to counter j, and put on the stack the fact that, when popped, it needs to be returned to counter k. Then, when we come to do the pop-transition corresponding to the split, we must first perform the pop-transitions corresponding to all the counters that were split by the constraints.

Remark 21. 1. Class memory functions are normally of the form $f : \mathcal{D} \to Q_\perp$. In our encoding we shall use a special set $\mathtt{Lab}$ of *labels* to keep track of local behaviour and will rely on functions $f : \mathcal{D} \to \mathtt{Lab}_\perp$ instead. Accordingly, our VPCMA will have a transition relation of the form

$$\Delta \subseteq Q \times \mathtt{Lab}_\perp \times (\Sigma_{\mathtt{push}} \cup \Sigma_{\mathtt{pop}}) \times \Gamma \times Q \times \mathtt{Lab} \;\cup\; Q \times \mathtt{Lab}_\perp \times \Sigma_{\mathtt{noop}} \times Q \times \mathtt{Lab}.$$

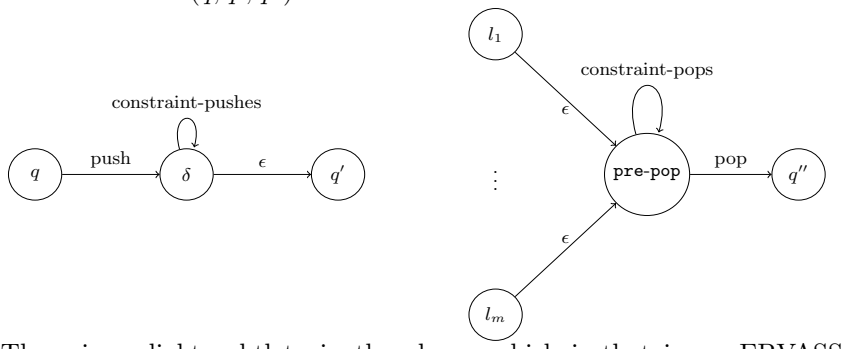
Note that the above can be easily accommodated by the standard definition by extending the set of states.

2. When we introduced EBVASS, we gave them the power to perform multiple increments and decrements in one transition. While this was useful in reducing VPCMA to EBVASS, we will now find it useful to simply permit a single increment or decrement in unary transitions, and decouple increments/decrements from split transitions.

In our reduction data values will be used to store the counter information. That is, the value of a counter will be represented by the number of data values that the class memory function assigns a label corresponding to that counter (we use the labels $1, \ldots, n$ for the n counters). When a counter is incremented, a fresh data value is read, and given the appropriate label. When a counter is

decremented, a data value with the label corresponding to that counter has its label changed to `done`. The fact that all increments have been decremented by the end of the run is then checked by the local acceptance condition.

To model constraints, after a push-transition corresponding to a split, we shall allow several more push-transitions corresponding to firings of the constraints. Firing a transition corresponding to a constraint (i, j, k) will take a data value with current label i, give it label j, and put a letter k on the stack. Then, when the corresponding pop-move is made, the label will be changed from `done` to k. The shape of the parts of the automaton corresponding to a split transition $\delta = (q, q', q'')$ is shown below



There is a slight subtlety in the above, which is that in an EBVASS all constraints are fired simultaneously at a split transition, not sequentially. Hence we should be sure that the same data value cannot be used to fire two constraints at the same split. Fortunately, this is already prevented, as if two such constraints were fired, when it came to make the corresponding pop-transitions, the first would fire correctly, but then the second could not because the data value would not have the `done`-label.

Thus, given an EBVASS $\mathcal{B} = (Q, q_0, L, n, \Delta_u, \Delta_s, C)$, we construct a SVPCMA $\mathcal{A}_\mathcal{B}$ as follows:

- The set of states of $\mathcal{A}_\mathcal{B}$ is $Q \uplus \Delta_s \uplus \{\texttt{pre-pop}\}$, where q_0 is initial;
- $\texttt{Lab} = \{1, \ldots, n\} \cup \{\texttt{done}, \texttt{split}\}$ and $\Gamma = Q \uplus \{1, \ldots, k\}$;
- $\Sigma = (\Delta_s \uplus C) + \{\texttt{split-pop}, \texttt{constraint-pop}\} + (\Delta_u \uplus \{\epsilon\})$;
- The set of globally accepting states is L;
- The set of locally accepting labels is $\{\texttt{done}\}$; and
- The transition relation is constructed as follows:
  - for each (unary) increment transition $\delta \in \Delta_u$ of the form $q \xrightarrow{+e_i} q'$ we have the transition $(q, \bot, \delta, q', i)$;
  - for each (unary) decrement transition $\delta \in \Delta_u$ of the form $q \xrightarrow{-e_i} q'$ we have the transition $(q, i, \delta, q', \texttt{done})$;
  - for each split transition $\delta \in \Delta_s$ of the form $q \to q' + q''$ we have the push-transition $(q, \bot, \delta, q'', \delta, \texttt{split})$, and the silent transition $\delta \xrightarrow{\epsilon} q'$;
  - for each $\delta \in \Delta_s$ and constraint $(i, j, k) \in C$ we have the push-transition $(\delta, i, (i, j, k), k, \delta, j)$;
  - for each $i \in \{1, \ldots, n\}$ we have the pop-transition $(\texttt{pre-pop}, \texttt{done}, \texttt{constraint-pop}, i, \texttt{pre-pop}, i)$;

- for each $l \in L$ we have the silent transition $l \xrightarrow{\epsilon} \texttt{pre-pop}$;
- finally, for each $q \in Q$ we have the pop-transition $(\texttt{pre-pop}, \texttt{split}, \texttt{split-pop}, q, q, \texttt{done})$.

Theorem 22. *The reachability problem for EBVASS is reducible to the emptiness problem for SVPCMA.*

8 Undecidability for $\mathsf{unit} \to \mathsf{unit} \to (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$

Here we show, by reduction from reachability in reset VASS [5], that observational equivalence in finitary RML is undecidable for closed terms of type $\mathsf{unit} \to \mathsf{unit} \to (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$. Observe that the arena used for modelling closed terms of this above has the following move structure: $q_0 \vdash a_0 \vdash q_1 \vdash a_1 \vdash q_2 \vdash a_2 \vdash q_3 \vdash a_3$ and $q_3 \vdash q_4 \vdash a_4$. Next we discuss how plays over the arena can be used to simulate reset VASS.

- The simulation will begin with $q_0\overleftarrow{a_0}\,\overleftarrow{q_1}\,\overleftarrow{a_1}\,\overleftarrow{q_2}\,\overleftarrow{a_2}\,\overleftarrow{q_3}\,\overleftarrow{q_4}$. This yields a play with pending questions q_3, q_4, which will block the formation of complete plays until the two questions are answered. We will take advantage of these questions at the very end of the simulation to check whether the simulation has reached an accepting state (if so, they will be answered).
- After the initialising segment discussed above, we shall have k segments $q_1\overleftarrow{a_1}$, where k is the number of counters. Each segment $q_1\overleftarrow{a_1}$ is used to represent a single counter and its identity as well as status (active or reset) will be stored in a local variable.
- Counter increments for counter j will be modelled with $q_2\overleftarrow{a_2}\,\overleftarrow{q_3}\,\overleftarrow{q_4}$, where q_2 is justified by the occurrence of a_1 corresponding to the jth counter. Each such segment will be equipped with a local variable that records the fact that the segment stores a singleton value of the relevant counter. The q_3q_4 moves are intended to contribute pending questions to the play (to create stack structure) and guarantee that a complete play can be formed only after the questions have been answered. In the final stage of the simulation, we shall use the need to answer these questions to check whether all increments have been matched by decrements (unless the counter has been reset in the meantime).
- Decrements will be represented by $q_3\overleftarrow{a_3}$, where q_3 is justified by a_2 from a segment corresponding to an (unreset) increment of the same counter. The local variable recording the singleton value will then be modified to reflect the fact that the value has been spent.
- Resets will be simulated by $q_2\overleftarrow{a_2}$, where q_2 is justified by q_1a_1 corresponding to the relevant counter. Its status will be updated to inactive and the q_1a_1 segment will not be used by the translation any more. However, in order to allow for further operations on the same counter, we shall create a new $q_1\overleftarrow{a_1}$ segment, which will be used as a target when simulating subsequent decrements.

– Zero testing, to be performed at the very end, will be triggered by O playing a_4 in response to the most recent q_4 used for modelling increments. If the corresponding q_3q_4 segment corresponds to a counter value that has been reset or decremented then a_3 will be played (otherwise the simulation will break - P will not respond). Finally, if all q_3q_4 corresponding to increments have been answered in this way, the first q_3q_4 segment will become pending. If O then plays a_4 then P will reply with a_3 iff the simulation has reached a final state.

Our main result is that, for any reset VASS, it is possible to build RML terms whose game semantics represents the reset VASS in the sense sketched above. This leads to:

Theorem 23. *Given a reset VASS* $\mathcal{A} = (Q, k, \Delta_i \cup \Delta_d \cup \Delta_0, q_0)$ *and target state* $q_f \in Q$*, there are* RML*-terms* $\vdash M, M' : \mathsf{unit} \to \mathsf{unit} \to (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$ *such that* $M \cong M'$ *iff there is a run of* $\mathcal{A}$ *reaching configuration* $(q_f, \bar{0})$.

As before, the only difference between M and M' is the place where final-state detection takes place: M will then terminate, wheereas M' will diverge.

Conclusion and Further Directions

For all types, we have a result giving the decidability status of a finitary RML fragment containing closed terms of that type, with the exception of the types in $\mathrm{RML}_{\mathrm{EBVASS}}$, for which we know observational equivalence is equivalent to EBVASS reachability. Clearly the open question of the decidability of EBVASS reachability, which seems interesting for its own sake, is especially important to us. More broadly, we do not yet have a complete classification of which types on the LHS of the turnstile give undecidability or decidability, nor a complete picture of which combinations of LHS and RHS types remain decidable. Settling these remaining questions would be a natural next step.

Acknowledgments. We are grateful to the anonymous reviewers for numerous constructive suggestions and to Ranko Lazić for discussions on VASS.

References

1. Abramsky, S., Jagadeesan, R., Malacaria, P.: Full abstraction for PCF. Inf. Comput. **163**(2), 409–470 (2000)
2. Abramsky, S., McCusker, G.: Linearity, sharing and state: a fully abstract game semantics for idealized Algol with active expressions. Electr. Notes Theor. Comput. Sci. **3**, 2–14 (1996)
3. Abramsky, S., McCusker, G.: Call-by-value games. In: Nielsen, M., Thomas, W. (eds.) CSL 1997. LNCS, vol. 1414, pp. 1–17. Springer, Heidelberg (1998). doi:10.1007/BFb0028004
4. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: STOC, pp. 202–211. ACM (2004)
5. Araki, T., Kasami, T.: Some decision problems related to the reachability problem for petri nets. Theor. Comput. Sci. **3**(1), 85–104 (1976)

6. Björklund, H., Schwentick, T.: On notions of regularity for data languages. In: Csuhaj-Varjú, E., Ésik, Z. (eds.) FCT 2007. LNCS, vol. 4639, pp. 88–99. Springer, Heidelberg (2007). doi:10.1007/978-3-540-74240-1_9
7. Björklund, H., Schwentick, T.: On notions of regularity for data languages. Theor. Comput. Sci. **411**(4–5), 702–715 (2010)
8. Cotton-Barratt, C.: Using class memory automata in algorithmic game semantics. Ph.D. thesis, University of Oxford (submitted, 2016)
9. Cotton-Barratt, C., Hopkins, D., Murawski, A.S., Ong, C.-H.L.: Fragments of ML decidable by nested data class memory automata. In: Pitts, A. (ed.) FoSSaCS 2015. LNCS, vol. 9034, pp. 249–263. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46678-0_16
10. Cotton-Barratt, C., Murawski, A.S., Ong, C.-H.L.: Weak and nested class memory automata. In: Dediu, A.-H., Formenti, E., Martín-Vide, C., Truthe, B. (eds.) LATA 2015. LNCS, vol. 8977, pp. 188–199. Springer, Cham (2015). doi:10.1007/978-3-319-15579-1_14
11. de Groote, P., Guillaume, B., Salvati, S.: Vector addition tree automata. In: LICS, pp. 64–73. IEEE Computer Society (2004)
12. Godlin, B., Strichman, O.: Regression verification. In: DAC, pp. 466–471. ACM (2009)
13. Honda, K., Yoshida, N.: Game-theoretic analysis of call-by-value computation. Theor. Comput. Sci. **221**(1–2), 393–456 (1999)
14. Hopkins, D.: Game semantics based equivalence checking of higher-order programs. Ph.D. thesis, Department of Computer Science, University of Oxford (2012)
15. Hopkins, D., Murawski, A.S., Ong, C.-H.L.: A fragment of ML decidable by visibly pushdown automata. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011. LNCS, vol. 6756, pp. 149–161. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22012-8_11
16. Hyland, J.M.E., Ong, C.-H.L.: On full abstraction for PCF: I, II, and III. Inf. Comput. **163**(2), 285–408 (2000)
17. Jacquemard, F., Segoufin, L., Dimino, J.: FO2($<+1, \sim$) on data trees, data tree automata and branching vector addition systems. Logical Methods Comput. Sci. **12**(2), 1–28 (2016)
18. Lazić, R., Murawski, A.S.: Contextual approximation and higher-order procedures. In: Jacobs, B., Löding, C. (eds.) FoSSaCS 2016. LNCS, vol. 9634, pp. 162–179. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49630-5_10
19. Lazic, R., Schmitz, S.: Nonelementary complexities for branching VASS, MELL, and extensions. ACM Trans. Comput. Log. **16**(3), 20:1–20:30 (2015)
20. Murawski, A.S.: Functions with local state: regularity and undecidability. Theor. Comput. Sci. **338**(1/3), 315–349 (2005)
21. Murawski, A.S., Ong, C.-H.L., Walukiewicz, I.: Idealized Algol with ground recursion, and DPDA equivalence. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 917–929. Springer, Heidelberg (2005). doi:10.1007/11523468_74
22. Murawski, A.S., Ramsay, S.J., Tzevelekos, N.: Game semantic analysis of equivalence in IMJ. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) ATVA 2015. LNCS, vol. 9364, pp. 411–428. Springer, Cham (2015). doi:10.1007/978-3-319-24953-7_30
23. Murawski, A.S., Tzevelekos, N.: Full abstraction for reduced ML. In: Alfaro, L. (ed.) FoSSaCS 2009. LNCS, vol. 5504, pp. 32–47. Springer, Heidelberg (2009). doi:10.1007/978-3-642-00596-1_4

24. Murawski, A.S., Tzevelekos, N.: Algorithmic nominal game semantics. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 419–438. Springer, Heidelberg (2011). doi:10.1007/978-3-642-19718-5_22
25. Murawski, A.S., Tzevelekos, N.: Algorithmic games for full ground references. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012. LNCS, vol. 7392, pp. 312–324. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31585-5_30
26. Pitts, A.M., Stark, I.D.B.: Operational reasoning for functions with local state. In: Higher Order Operational Techniques in Semantics, pp. 227–273. Cambridge University Press (1998)
27. Reynolds, J.C.: The essence of ALGOL. In: Proceedings of the International Symposium on Algorithmic Languages. Elsevier Science Inc. (1981)