ML, Visibly Pushdown Class Memory Automata, and Extended Branching Vector Addition Systems with States

CONRAD COTTON-BARRATT, Jump Trading London
ANDRZEJ S. MURAWSKI, University of Oxford
C.-H. LUKE ONG, University of Oxford

We prove that the observational equivalence problem for a finitary fragment of the programming langauge ML is recursively equivalent to the reachability problem for *extended branching vector addition systems with states* (EBVASS). This result has two natural and independent parts. We first prove that the observational equivalence problem is equivalent to the emptiness problem for a new class of class memory automata equipped with a visibly pushdown stack, called *Visibly Pushdown Class Memory Automata* (VPCMA). Our proof uses the fully abstract game semantics of the language. We then prove that the VPCMA emptiness problem is equivalent to the reachability problem for 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.

CCS Concepts: • **Theory of computation** → **Program verification**; *Automata over infinite objects*; *Denotational semantics*; • **Software and its engineering** → Petri nets;

Additional Key Words and Phrases: Higher-Order Types, Game Semantics, Full Abstraction, Vector Addition Systems, Automata over Infinite Alphabets

ACM Reference format:
Conrad Cotton-Barratt, Andrzej S. Murawski, and C.-H. Luke Ong. 0. ML, Visibly Pushdown Class Memory Automata, and Extended Branching Vector Addition Systems with States. *ACM Trans. Program. Lang. Syst.* 0, 0, Article 0 (0), 39 pages.
https://doi.org/0000001.0000001

Contents

This work was done when the author was at the University of Oxford.
This is an extended and revised version of a paper that appeared in ESOP'17 [Cotton-Barratt et al. 2017]. An important difference is that the fragment of ML called RML$_{\text{EBVASS}}$ in *op. cit.* is renamed RML$_{\text{VPC}}$ in this paper.

0164-0925/0/0-ART0 $15.00
https://doi.org/0000001.0000001

Shape	LHS Type, θ_i	RHS Type, θ
I [Cotton-Barratt et al. 2015a]	$(\beta \to \beta) \to \cdots \to (\beta \to \beta) \to \beta$	$\beta \to \cdots \to \beta$
II [Hopkins et al. 2011]	$((\beta \to \cdots \to \beta) \to \beta) \to \cdots \to ((\beta \to \cdots \to \beta) \to \beta) \to \beta$	$(\beta \to \cdots \to \beta) \to \beta$

Fig. 1. Two decidable fragments of finitary RML

1 INTRODUCTION

RML is a prototypical call-by-value functional language with state [Abramsky and McCusker 1997], 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 [Reynolds 1981] – the call-by-name counterpart of RML, the decidability of observational equivalence depends on the type-theoretic order [Murawski et al. 2005] 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 [Murawski 2005], 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 ($i = 1, 2$)

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

from $\mathcal{L}$, are they observationally equivalent? Unsurprisingly, the general problem is undecidable [Murawski 2005]. 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] 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 [Cotton-Barratt et al. 2015a] 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

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

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

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) [Jacquemard et al. 2016], whose decidability status is, to our knowledge, unknown. Our result applies not only to closed terms but also to the fragment $\mathrm{RML}_{\mathrm{VPC}}$ (Theorem 2.2) of open terms of type (2) in which free variables are subject to certain type constraints. Our main result is the following

Theorem 1.1. *Observational equivalence for the terms-in-context in* $\mathrm{RML}_{\mathrm{VPC}}$ *is recursively equivalent to the reachability problem for extended branching vector addition systems.*

Our second result (Theorem 9.2) is that the reachability problem for *reset vector addition systems with states* [Araki and Kasami 1976] 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 [Cotton-Barratt et al. 2015a; Hopkins et al. 2011], Theorem 1.1 and Theorem 9.2 are proved by appealing to the game semantics for RML [Abramsky and McCusker 1997; Honda and Yoshida 1999], which is *fully abstract*, i.e., the equational theory induced by the semantics coincides with observational equivalence. In game semantics [Abramsky et al. 2000; Hyland and Ong 2000], 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 [Abramsky and McCusker 1997; Honda and Yoshida 1999], 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 [Ghica and McCusker 2000; Ong 2002]. In our case, the main technical challenge lies in the encoding of the justification pointers of the plays.

In recent work [Cotton-Barratt 2017; Cotton-Barratt et al. 2015a], 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) [Cotton-Barratt et al. 2015b], which are a variant of *class memory automata* [Björklund and Schwentick 2007] 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 $\mathrm{RML}_{\mathrm{VPC}}$ may be viewed as an interleaving of "visibly pushdown" threads, subject to the global well-bracketing condition. (See Section 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 [Cotton-Barratt et al. 2015a]: 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 RML_{VPC} term-in-context are representable as a weak deterministic SVPCMA (Theorem 6.2). Thanks to the closure property of SVPCMA, it then follows that RML_{VPC} observational equivalence is reducible to the emptiness problem for VPCMA (Theorem 6.1).

Finally and most importantly, we show (Theorem 8.3 and Theorem 8.7) that the emptiness problem for VPCMA (equivalently for SVPCMA) is equivalent to the reachability problem for extended branching VASS (EBVASS) [Jacquemard et al. 2016], 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) [de Groote et al. 2004], which is known to be non-elementary [Lazic and Schmitz 2015].

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	unit → · · · → unit	NDCMA / decidable [Cotton-Barratt et al. 2015a; Hopkins 2012]
2	(unit → · · · → unit) → unit	VPA / decidable [Hopkins et al. 2011]
2	unit → (unit → · · · → unit) → unit	EBVASS (this paper)
2	unit → unit → (unit → unit) → unit	Undecidable (this paper)
2	(unit → unit) → unit → unit	Undecidable [Cotton-Barratt et al. 2015a]
3	((unit → unit) → unit) → unit	Undecidable [Cotton-Barratt et al. 2015a]

Related work. Hopkins and Murawski [Hopkins 2012] 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 [Cotton-Barratt et al. 2015a]. Automata over an infinite alphabet (specifically, pushdown register automata) have also been applied to game semantics [Murawski and Tzevelekos 2011, 2012] for a different purpose, namely, to model generation of fresh names in fragments of ML [Murawski and Tzevelekos 2012] and Java [Murawski et al. 2015]. When

[3] *Weak* class memory automata [Cotton-Barratt 2017; Cotton-Barratt et al. 2015a] are class memory automata in which the local acceptance condition is dropped.

extended with name storage, observational equivalence of terms-in-context with types in $\mathrm{RML}_{\mathrm{VPC}}$ becomes undecidable [Murawski and Tzevelekos 2012]; in particular, this is already the case for closed terms of type unit → unit → unit.

Organisation of the paper

We first present an outline of the paper. Then we provide guidance on how this paper should be read.

Outline. In Section 2 we define the syntax and operational semantics of RML and the fragment $\mathrm{RML}_{\mathrm{VPC}}$. In Section 3 we present the game semantics for RML. We then explain, in Section 4, why an automaton over an infinite alphabet augmented with a visibly pushdown stack is needed to model plays of the $\mathrm{RML}_{\mathrm{VPC}}$-terms. The automata models, VPMCA and SVPCMA, are presented in Section 5, where we show that their emptiness problems are interreducible, and discuss their closure properties. In Section 6 we show that the complete plays in the game semantics of $\mathrm{RML}_{\mathrm{VPC}}$-terms are representable as weak deterministic SVPCMA. Consequently the observational equivalence of $\mathrm{RML}_{\mathrm{VPC}}$-terms is reducible to the emptiness problem of SVPCMA (and equivalently to that of VPCMA). Reducibility in the opposite direction is then shown in Section 7. In Section 8 we introduce EBVASS and show that its reachability problem and the emptiness problem for VPCMA are interreducible. Finally, in Section 9, we show that observational equivalence for closed terms of type unit → unit → (unit → unit) → unit is undecidable.

How to read this paper. This paper has two main results:

(1) Observational equivalence of $\mathrm{RML}_{\mathrm{VPC}}$ is recursively equivalent to (S)VPCMA emptiness (Section 2 to 7)
(2) (S)VPCMA emptiness is recursively equivalent to EBVASS reachability (Section 8).

Readers interested in (algorithmic) game semantics should read the paper as it has been presented. Result (2) is a contribution, in its own right, to the algorithmic foundations of verification in general and automata over data words / trees in particular. Readers interested in this automata-theoretic result need only read Sections 5 and 8 which are self-contained.

2 A STATEFUL CALL-BY-VALUE FUNCTIONAL LANGUAGE RML

RML is a call-by-value functional language with state [Abramsky and McCusker 1997]. 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 → operator:

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

The *order* of a type is given by

$$\begin{array}{rcl} \mathbf{ord}(\mathsf{int}) & = & 0, \\ \mathbf{ord}(\mathsf{unit}) & = & 0, \\ \mathbf{ord}(\mathsf{int\ ref}) & = & 1, \\ \mathbf{ord}(\theta \to \theta') & = & max(\mathbf{ord}(\theta) + 1, \mathbf{ord}(\theta')). \end{array}$$

In order to eliminate obvious sources of undecidability, we consider *finitary* RML, with finite ground types (int = $\{0, \cdots, \mathrm{max}\}$), 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

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

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

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

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

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

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

Fig. 2. Syntax of finitary RML

we work with finite int. We will write **let** $x = M$ **in** N as syntactic sugar for $(\lambda x.N)M$, and $M; N$ for **let** $x = M$ **in** N where x is chosen to be fresh in N.

The operational semantics of the language is presented as a "big-step" relation that uses *stores* [Abramsky and McCusker 1997] 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, $\mathbf{mkvar}(V_1, V_2)$.

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 Theorem 2.1. OE is a natural notion of program equivalence, a key problem in verification [Godlin and Strichman 2009].

Definition 2.1. 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*, written $M \cong N$, if for all contexts $C[-]$ such that $\vdash C[M], C[N] : \mathsf{unit}$, $C[M] \Downarrow$ iff $C[N] \Downarrow$.

Remark 2.1.1. RML is similar to Reduced ML [Pitts and Stark 1998], the restriction of Standard ML to ground-type references, but is augmented with a "bad-variable" constructor in the sense of Reynolds [Reynolds 1981]. **mkvar** makes it possible to create terms of type int ref with non-standard ("bad") behaviour. For example, $\mathbf{mkvar}(\lambda x^{\mathsf{unit}}.1, \lambda y^{\mathsf{int}}.\mathbf{while}\ 1\ \mathbf{do}\ ())$ is a variable that will return 1 when dereferenced and the first attempt to write to it will result in divergence. Because of the

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

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

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

$$\frac{s, M \Downarrow s', \mathbf{mkvar}(V_0, V_1) \qquad s', V_0() \Downarrow s'', V}{s, !M \Downarrow s'', V}$$

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

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

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

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

Fig. 3. Operational semantics of RML

constructor, the equality test on variables does not quite make sense any more. However, in its absence, such a test is definable.

In the presence of 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 int ref are positive. The semantics of 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 [Murawski and Tzevelekos 2009].

Definition 2.2. The fragment $\mathrm{RML}_{\mathrm{VPC}}$ 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

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

Example 2.3. The following term $\vdash M : \mathsf{int} \to (\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$ is in $\mathrm{RML}_{\mathrm{VPC}}$.

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

```
let y=f(x) in (stop:=1; y)
```

We write **assert**(M) for **if** M **then** () **else** Ω, where Ω is the divergent term **while** 1 **do** (). When applied to an integer x, the term yields a function of type (int $\rightarrow$ int) $\rightarrow$ int, which will apply its argument (a function f : int $\rightarrow$ int) to x. However, owing to the assertion and the side effects, the behaviour of $M\,x$ is quite different from $\lambda f^{\mathsf{int}\rightarrow\mathsf{int}}.f\,x$. If x is even then only sequential (non-overlapping) uses of the function will be allowed. Thus, **let** $g = M\,0$ **in** (**let** $a = g(\lambda x^{\mathsf{int}}.0)$ **in** $g(\lambda y^{\mathsf{int}}.0)$) terminates, whereas **let** $g = M\,0$ **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, **let** $g = M\,1$ **in** $g(\lambda x^{\mathsf{int}}.g(\lambda y^{\mathsf{int}}.0))$ terminates, whereas **let** $g = M\,1$ **in** (**let** $a = g(\lambda x^{\mathsf{int}}.0)$ **in** $g(\lambda y^{\mathsf{int}}.0)$) diverges. The term M is not observationally equivalent to $\lambda x^{\mathsf{int}}.\lambda f^{\mathsf{int}\rightarrow\mathsf{int}}.f\,x$, but it is equivalent to the term given below.

```
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)
```

3 GAME SEMANTICS OF RML

We use a presentation of call-by-value game semantics in the style of Honda and Yoshida [Honda and Yoshida 1999], as opposed to Abramsky and McCusker's isomorphic model [Abramsky and McCusker 1997], 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 [Hopkins et al. 2011].

Definition 3.1. An *arena* A is a tuple $(M_A, I_A, \vdash_A, \lambda_A)$, where

- M_A is a set of *moves*,
- $I_A \subseteq M_A$ contains *initial* moves,
- $\vdash_A \subseteq M_A \times (M_A \backslash I_A)$ is called the *enabling relation*, and
- $\lambda_A : M_A \longrightarrow \{O, P\} \times \{Q, A\}$ 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 $(\pi_2 \circ \lambda_A)(m') = A \Rightarrow (\pi_2 \circ \lambda_A)(m) = Q$. We write π_1, π_2 to refer to the two projections from $\{O, P\} \times \{Q, A\}$.

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 shall use q and a to range over questions and answers respectively.

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 $\star$), and $\mathbb{Z}$ (has integers as moves, all of which are initial P-answers). In all these cases, the enabling relation is empty. The constructions on arenas are defined in Fig. 4 and Fig. 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, $A \longrightarrow B$,

$$\begin{array}{rcl}
M_{A\otimes B} & = & I_A \times I_B \uplus \overline{I_A} \uplus \overline{I_B} \\
I_{A\otimes B} & = & I_A \times I_B \\
\lambda_{A\otimes B} & = & m \mapsto \begin{cases} PA & \text{if } m \in I_A \times I_B \\ \lambda_A(m) & \text{if } m \in \overline{I_A} \\ \lambda_B(m) & \text{if } m \in \overline{I_B} \end{cases} \\
\vdash_{A\otimes B} & = & \{\, ((i_A, i_B), m) \mid i_A \in I_A \wedge i_B \in I_B \;\wedge\; (i_A \vdash_A m \vee i_B \vdash_B m) \,\} \\
 & & \cup (\vdash_A \; \cap (\overline{I_A} \times \overline{I_A})) \;\cup\; (\vdash_B \; \cap (\overline{I_B} \times \overline{I_B}))
\end{array}$$

$$\begin{array}{rcl}
M_{A\Rightarrow B} & = & \{\bullet\} \uplus M_A \uplus M_B \\
I_{A\Rightarrow B} & = & \{\bullet\} \\
\lambda_{A\Rightarrow B} & = & m \mapsto \begin{cases} PA & \text{if } m = \bullet \\ OQ & \text{if } m \in I_A \\ \overline{\lambda_A}(m) & \text{if } m \in \overline{I_A} \\ \lambda_B(m) & \text{if } m \in M_B \end{cases} \\
\vdash_{A\Rightarrow B} & = & \{\, (\bullet, i_A) \mid i_A \in I_A \} \;\cup\; \{\, (i_A, i_B) \mid i_A \in I_A, i_B \in I_B \} \cup \vdash_A \; \cup \vdash_B
\end{array}$$

$$\begin{array}{rcl}
M_{A\to B} & = & M_A \uplus M_B \\
I_{A\to B} & = & I_A \\
\lambda_{A\to B}(m) & = & \begin{cases} OQ & \text{if } m \in I_A \\ \overline{\lambda_A}(m) & \text{if } m \in \overline{I_A} \\ \lambda_B(m) & \text{if } m \in M_B \end{cases} \\
\vdash_{A\to B} & = & \{\, (i_A, i_B) \mid i_A \in I_A,\; i_B \in I_B \} \cup \vdash_A \; \cup \vdash_B
\end{array}$$

Fig. 4. Arena and prearena constructions: definitions

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 $\bullet$.

We intend arenas to represent types. In particular

$$\begin{array}{rcl}
[\![\text{unit}]\!] & = & 1, \\
[\![\text{int}]\!] & = & \mathbb{Z} \text{ (or } \{0, \cdots, \text{max}\} \text{ for finitary RML),} \\
[\![\text{int ref}]\!] & = & [\![\text{unit} \to \text{int}]\!] \otimes [\![\text{int} \to \text{unit}]\!], \\
[\![\theta_1 \to \theta_2]\!] & = & [\![\theta_1]\!] \Rightarrow [\![\theta_2]\!].
\end{array}$$

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 following standard conditions [Abramsky and McCusker 1997]:

(i) *Alternation*: O and P take it in turns to make a move. That is if $t\, m_1\, m_2 \sqsubseteq s$ then $\lambda^{OP}(m_1) \neq \lambda^{OP}(m_2)$.

(ii) *Well-Bracketing*: Questions asked first must be answered first. If $t_1\, \overset{\curvearrowleft}{q\, t_2\, a} \sqsubseteq s$ then all questions in t_2 must be answered in t_2.

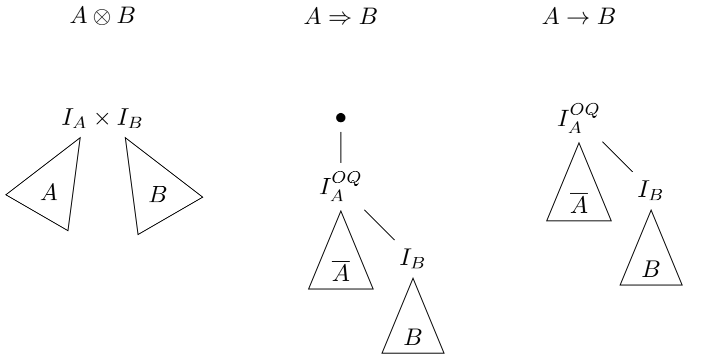


Fig. 5. Arena and prearena constructions, pictorially

(iii) *Visibility*: If $t_1\, \overset{\curvearrowleft}{m_1\, t_2\, m_2} \sqsubseteq s$ then m_1 appears in $\mathsf{view}(t_1\, m_1\, t_2)$, where view is defined by: $\mathsf{view}(\epsilon) = \epsilon$, $\mathsf{view}(o) = o$ if o is initial, and $\mathsf{view}(t_1\, \overset{\curvearrowleft}{m_1\, t_2\, m_2}) = \mathsf{view}(t_1)\, \overset{\curvearrowleft}{m_1\, m_2}$.

We denote the set of all valid plays over prearena A as P_A.

Definition 3.2. 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. For instance, the *identity strategy* $id_A : A \longrightarrow A$ always tells P to respond with the same move as the one just played by O, but in the opposite copy of A:

$$id_A = \{ s \in P_A \mid s \restriction L = s \restriction R \}$$

where $s \restriction L$ (resp. $s \restriction R$) restrict s to moves from the left (resp. right) copy of A in $A \longrightarrow A$. *Projection strategies* $\pi^1_{A,B} : A \otimes B \longrightarrow A$ work in a similar way except that the B-move used in the initial move is ignored and the strategy replies to the initial move just by copying its A-component to the right copy of A. Afterwards, the strategy behaves in the same way as the identity strategy and no further moves from B occur during the play. The projection strategies $\pi^2_{A,B} : A \otimes B \longrightarrow B$ are defined analogously by ignoring A.

When giving examples, we shall often resort to subscripts to indicate the origin of moves, i.e. whenever an arena is labelled with a subscript X, moves originating from it will be written m_X. For instance, the *conditional strategy* $if_A : [\![\mathsf{int}]\!] \otimes (1 \Rightarrow A)_{L_1} \otimes (1 \Rightarrow A)_{L_2} \longrightarrow A$, after the initial move $(j, \bullet_{L_1}, \bullet_{L_2})$, responds with $\star_{L_1}$ or $\star_{L_2}$, depending on whether j is different from 0 or not. Once $\star_{L_k}$ is played ($k = 1, 2$), the next move must be an initial move of the copy of A tagged L_k and the strategy then turns into the identity strategy between A_{L_k} and the right copy of A. Formally, we have

$$if_A = \{ \overset{\curvearrowleft}{(j, \bullet_{L_1}, \bullet_{L_2})\ \star_{L_k}}\ \ s \mid (j \neq 0\ \wedge k = 1) \vee (j = 0\ \wedge\ k = 2),\ \ s \restriction L_k = s \restriction R \}.$$

Example 3.3. In Figure 6 we give further examples of strategies used to model RML by listing their maximal plays using shorthands for regular expressions. Note how the *ref* strategy corresponds to the behaviour of a memory cell: whenever O plays $\star_{R_1}$, the response depends on the last k_{R_2} move and, if no such move has occurred yet, the response j_{R_1} corresponds to the initial value used in j_L.

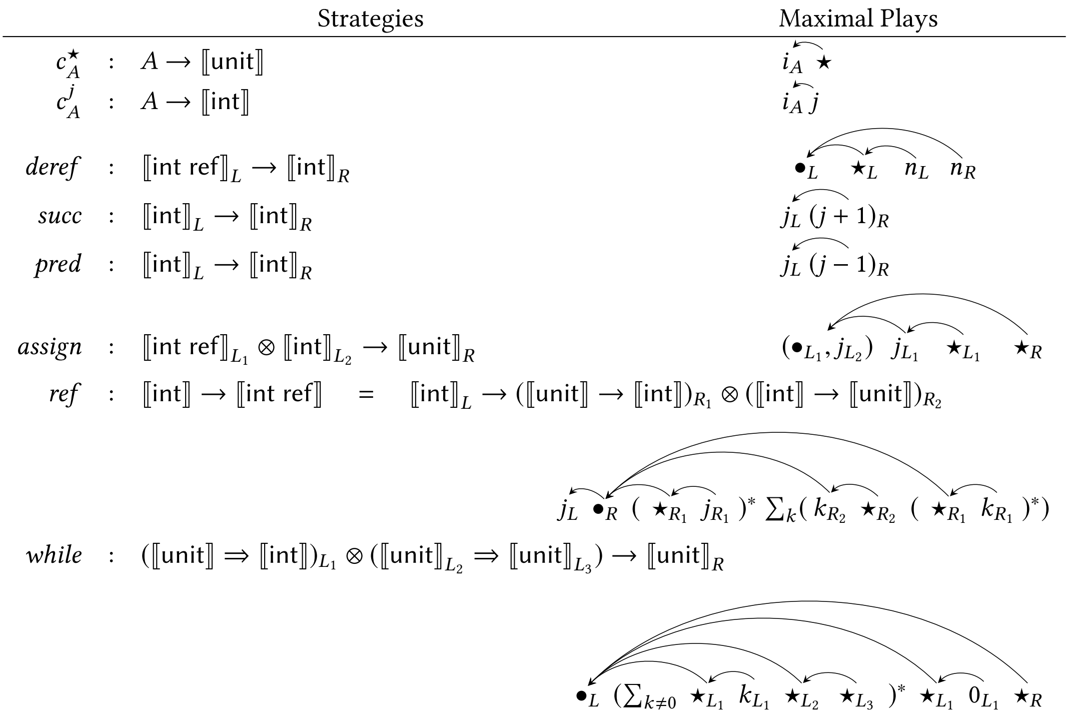

Strategies
Maximal Plays
$c_A^\star : A \longrightarrow [\![\mathsf{unit}]\!]$
$i_A \; \star$
$c_A^j : A \longrightarrow [\![\mathsf{int}]\!]$
$i_A \; j$
$\mathit{deref} : [\![\mathsf{int\ ref}]\!]_L \longrightarrow [\![\mathsf{int}]\!]_R$
$\bullet_L \; \star_L \; n_L \; n_R$
$\mathit{succ} : [\![\mathsf{int}]\!]_L \longrightarrow [\![\mathsf{int}]\!]_R$
$j_L \; (j+1)_R$
$\mathit{pred} : [\![\mathsf{int}]\!]_L \longrightarrow [\![\mathsf{int}]\!]_R$
$j_L \; (j-1)_R$
$\mathit{assign} : [\![\mathsf{int\ ref}]\!]_{L_1} \otimes [\![\mathsf{int}]\!]_{L_2} \longrightarrow [\![\mathsf{unit}]\!]_R$
$(\bullet_{L_1}, j_{L_2}) \; j_{L_1} \; \star_{L_1} \; \star_R$
$\mathit{ref} : [\![\mathsf{int}]\!] \longrightarrow [\![\mathsf{int\ ref}]\!] = [\![\mathsf{int}]\!]_L \longrightarrow ([\![\mathsf{unit}]\!] \longrightarrow [\![\mathsf{int}]\!])_{R_1} \otimes ([\![\mathsf{int}]\!] \longrightarrow [\![\mathsf{unit}]\!])_{R_2}$
$j_L \; \bullet_R \; (\; \star_{R_1} \; j_{R_1} \;)^* \; \sum_k (\; k_{R_2} \; \star_{R_2} \; (\; \star_{R_1} \; k_{R_1} \;)^*)$
$\mathit{while} : ([\![\mathsf{unit}]\!] \Rightarrow [\![\mathsf{int}]\!])_{L_1} \otimes ([\![\mathsf{unit}]\!]_{L_2} \Rightarrow [\![\mathsf{unit}]\!]_{L_3}) \longrightarrow [\![\mathsf{unit}]\!]_R$
$\bullet_L \; (\sum_{k \neq 0} \; \star_{L_1} \; k_{L_1} \; \star_{L_2} \; \star_{L_3} \;)^* \; \star_{L_1} \; 0_{L_1} \; \star_R$

Fig. 6. Examples of strategies

As detailed in Figure 7, strategies are assigned to terms by induction on the syntax using several operations discussed below, notably, strategy composition.

Composition

The composite $\sigma; \tau : A \longrightarrow C$ of two given strategies $\sigma : A \longrightarrow B$ and $\tau : B \longrightarrow C$ is defined by letting σ and τ interact through the common component B, and subsequently erasing the communications in B.

To capture the interactions formally, we first introduce an auxiliary prearena that will accommodate the interactions: given arenas A, B, C, let (A, B, C) be the prearena obtained by setting

$$\begin{aligned}
M_{(A,B,C)} &= M_{A \to B} \uplus M_C, \\
I_{(A,B,C)} &= I_A \\
\lambda_{(A,B,C)} &= [\lambda_{A \to B}[I_B \mapsto PQ], \overline{\lambda}_C] \\
\vdash_{(A,B,C)} &= \vdash_{A \to B} \cup \{ (i_B, i_C) \mid i_B \in I_B, \; i_C \in I_C \} \cup \vdash_C
\end{aligned}$$

Let u be a justified sequence on (A, B, C). We define $u \upharpoonright BC$ to be u in which all A-moves are suppressed. $u \upharpoonright AB$ is defined analogously. $u \upharpoonright AC$ is defined similarly with the caveat that, if there was a pointer from an initial C-move to an initial B-move, which in turn had a pointer to an A-move, we add a pointer from the C-move to the A-move. u will be called an *interaction sequence* on A, B, C if $u \upharpoonright AB \in P_{A \to B}$, $u \upharpoonright BC \in P_{B \to C}$ and $u \upharpoonright AC \in P_{A \to C}$. We write $\mathrm{inter}(A, B, C)$ for the set of interaction sequences on A, B, C.

Definition 3.4. Given $\sigma : A \longrightarrow B$ and $\tau : B \longrightarrow C$, we let

$$\sigma;\tau : A \longrightarrow C = \{\, u \upharpoonright AC \mid u \in \mathsf{inter}(A, B, C),\ u \upharpoonright AB \in \sigma,\ u \upharpoonright BC \in \tau \,\}.$$

As can be seen in Figure 7, several term constructors can be interpreted simply by composition with suitable special strategies.

The categorical structure allowing for the interpretation of RML types is discussed in [Honda and Yoshida 1999]. Here we review the shape of two most significant operations: pairing and currying.

Pairing

The (left) pairing operation combines two strategies $\sigma : C \longrightarrow A$ and $\tau : C \longrightarrow B$ into a single strategy $\langle\sigma, \tau\rangle : C \longrightarrow A \otimes B$. $\langle\sigma, \tau\rangle$ initially behaves in the same way as σ but only until σ is about to play an initial move i_A in A. Then that move is not played by $\langle\sigma, \tau\rangle$ – instead it starts behaving as τ as long as τ does not play in B. If τ is about to play i_B in B, $\langle\sigma, \tau\rangle$ will play (i_A, i_B) in $A \otimes B$. Afterwards, $\langle\sigma, \tau\rangle$ amounts to an interleaving of the remaining behaviours in σ and τ, depending on whether O plays a move justified by i_A or i_B. Intuitively, the strategy mimics left-to-right evaluation of two components and their subsequent interaction. Clearly, the pairing construction can be extended to an arbitrary number of strategies. In giving the semantics of terms, pairing is used to group together strategies corresponding to subterms before composition with a special strategy.

Currying

In our model we do not have the call-by-name 1-1 correspondence between strategies from $C \otimes A \longrightarrow B$ and $C \longrightarrow (A \Rightarrow B)$. However, it can be recovered partially [Honda and Yoshida 1999], which suffices for the call-by-value interpretation.

Given $\sigma : C \otimes A \longrightarrow B$, the strategy $p\lambda(\sigma) : C \longrightarrow (A \Rightarrow B)$ is defined by responding to the initial move i_C with $\bullet$ (from $A \Rightarrow B$). From then on, each time O plays an initial move i_A from A, $p\lambda(\sigma)$ behaves as a fresh copy of σ would after (i_C, i_A). As a consequence, $p\lambda(\sigma)$ consists of independent interleaved copies of σ, each of which is determined by an initial move in A. Strategies that exhibit this uniform kind of behaviour are known to be in 1-1 correspondence with strategies on $C \otimes A \longrightarrow B$. For example, the strategy $ev_{A,B} : (A \Rightarrow B) \otimes A \longrightarrow B$ used to interpret application can be obtained by taking $(p\lambda)^{-1}(id_{A \Rightarrow B})$. Accordingly, it will copy moves between the two instances of B and the two instances of A. The latter corresponds to communication between the argument and the function.

Given $\sigma : A \longrightarrow B$, we define $\sigma_\perp = p\lambda(\pi_{A,1}; \sigma) : A \longrightarrow (1 \Rightarrow B)$. Intuitively, $\sigma_\perp$ amounts to a thunk $1 \Rightarrow B$ that calls σ on each application. The construction is useful in that it delays the actions of σ. In particular, it will be used to model the lack of immediate evaluation for conditional branches.

Full abstraction

In the game model of RML, a term-in-context $\Gamma \vdash M : \theta$, where $\Gamma = \{x_1 : \theta_1, \ldots, x_n : \theta_n\}$, is interpreted by a strategy of the prearena $[\![\Gamma]\!] \longrightarrow [\![\theta]\!]$, where $[\![\Gamma]\!]$ stands for $[\![\theta_1]\!] \otimes \ldots \otimes [\![\theta_n]\!]$. These strategies are defined by recursion over the syntax of the term, as shown in Figure 7.

Remark 3.4.1.

- *Free identifiers $\Gamma, x : \theta \vdash x : \theta$ are interpreted as* copy-cat *strategies, where P always copies O's move into the other copy of $[\![\theta]\!]$.*
- *Application MN is interpreted by composition with the relevant instance of ev, which synchronises the strategy corresponding to the argument with the strategy corresponding to the function on moves in $[\![\theta]\!]$.*
- *$\lambda x.M$ allows multiple copies of $[\![M]\!]$ to be run in parallel, subject to the interleavings being a valid play.*

$$
\begin{array}{rcl}
\llbracket \Gamma \vdash () \rrbracket & = & c^{\star}_{\llbracket \Gamma \rrbracket} \\
\llbracket \Gamma \vdash \mathrm{j} \rrbracket & = & c^{j}_{\llbracket \Gamma \rrbracket} \\
\llbracket \Gamma, x : \theta \vdash x : \theta \rrbracket & = & \pi^{2}_{\llbracket \Gamma \rrbracket, \llbracket \theta \rrbracket} \\
\llbracket \Gamma \vdash \mathbf{succ}(M) \rrbracket & = & \llbracket \Gamma \vdash M \rrbracket; \mathit{succ} \\
\llbracket \Gamma \vdash \mathbf{pred}(M) \rrbracket & = & \llbracket \Gamma \vdash M \rrbracket; \mathit{pred} \\
\llbracket \Gamma \vdash \mathbf{if}\, M \,\mathbf{then}\, M_1 \,\mathbf{else}\, M_0 \rrbracket & = & \langle \llbracket \Gamma \vdash M \rrbracket, \llbracket \Gamma \vdash M_1 \rrbracket_{\perp}, \llbracket \Gamma \vdash M_0 \rrbracket_{\perp} \rangle; \mathit{if}_{\llbracket \theta \rrbracket} \\
\llbracket \Gamma \vdash !M \rrbracket & = & \llbracket \Gamma \vdash M \rrbracket; \mathit{deref} \\
\llbracket \Gamma \vdash M := N \rrbracket & = & \langle \llbracket \Gamma \vdash M \rrbracket, \llbracket \Gamma \vdash N \rrbracket \rangle; \mathit{assign} \\
\llbracket \Gamma \vdash \mathbf{ref}\, M \rrbracket & = & \llbracket \Gamma \vdash M \rrbracket; \mathit{ref} \\
\llbracket \Gamma \vdash \mathbf{mkvar}(M, N) \rrbracket & = & \langle \llbracket \Gamma \vdash M \rrbracket, \llbracket \Gamma \vdash N \rrbracket \rangle \\
\llbracket \Gamma \vdash \mathbf{while}\, M \,\mathbf{do}\, N \rrbracket & = & \langle \llbracket \Gamma \vdash M \rrbracket_{\perp}, \llbracket \Gamma \vdash N \rrbracket_{\perp} \rangle; \mathit{while} \\
\llbracket \Gamma \vdash MN \rrbracket & = & \langle \llbracket \Gamma \vdash M \rrbracket, \llbracket \Gamma \vdash N \rrbracket \rangle; \mathit{ev}_{\llbracket \theta \rrbracket, \llbracket \theta' \rrbracket} \\
\llbracket \Gamma \vdash \lambda x^{\theta}.M \rrbracket & = & p\lambda(\llbracket \Gamma, x : \theta \vdash M \rrbracket)
\end{array}
$$

Fig. 7. The game semantics of RML

- *Variable binding,* **let** x = **ref** 0 **in** M*, corresponds to* $(\lambda x.M)(\mathbf{ref}\ 0)$*, which will synchronise the strategy for M with the cell strategy restricting M to "good-variable" scenarios. Subsequently, the communication with the cell strategy will be erased, as in the definition of composition.*

The game-semantic model is fully abstract in the following sense. 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 $\mathbf{comp}(\sigma)$.

Theorem 3.5. *[Abramsky and McCusker 1996, 1997] If* $\Gamma \vdash M : \theta$ *and* $\Gamma \vdash N : \theta$ *are* RML *terms then* $\Gamma \vdash M \cong N$ *iff* $\mathbf{comp}(\llbracket M \rrbracket) = \mathbf{comp}(\llbracket N \rrbracket)$.

4 ANALYSIS OF SHAPE OF PLAYS FOR RML$_{\mathrm{VPC}}$

Theorem 3.5 makes it possible to recast contextual equivalence as language equivalence, provided we find a way to represent $\mathbf{comp}(\llbracket M \rrbracket)$ and $\mathbf{comp}(\llbracket N \rrbracket)$ as languages. In previous work, certain fragments of RML have been captured in this way using automata-theoretic formalisms such as finite-state machines [Murawski 2005], visibly pushdown automata [Hopkins et al. 2011] and nested data class memory automata [Cotton-Barratt et al. 2015a].

In order to capture the game semantics of RML$_{\mathrm{VPC}}$, in this paper we introduce a new kind of automaton over an infinite alphabet which is equipped with a visibly pushdown stack. Below we highlight why these features are desirable when trying to capture plays generated by RML$_{\mathrm{VPC}}$ terms. The shape of the prearenas for terms-in-context in this fragment is shown in Fig. 8.

Meaning of moves

We describe the intuitive meaning of various moves from the Figure. The question q_0 starts the evaluation of the term. The answer a_0 signals that the evaluation (to a function value of type $\beta \to (\beta \to \cdots \to \beta) \to \beta$) was successful. Then q_1 corresponds to calling 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 returned. The question q_2 then corresponds to calling the value on a function argument (of type $\beta_1 \to \cdots \to \beta_m$), while $q^1_*, a^1_*, .., q^{m-1}_*, a^{m-1}_*$ represent interaction with that argument. Finally, a_2 means that the call has returned. Note that $q^1_*, a^1_*, q^{m-1}_*, a^{m-1}_*$ correspond to interacting with a value of type $\beta_1 \to \cdots \to \beta_m$. Consequently, q^1_*, a^1_* represent a call/return pair that generates a value

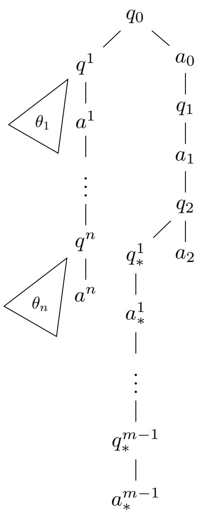


Fig. 8. Shape of prearena for $\theta_1 \to \cdots \to \theta_n \to \beta \vdash \beta \to (\beta_1 \to \cdots \to \beta_m) \to \beta$

of type $\beta_2 \to \cdots \to \beta_m$. More generally, q_*^i $(1 \le i \le m-1)$ refers to calling a function of type $\beta_i \to \cdots \to \beta_m$, while a_*^i signals a return value of type $\beta_{i+1} \to \cdots \to \beta_m$. Thus, a_*^{m-1} corresponds to a value of type β_m.

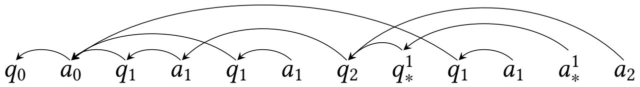
Anatomy of complete plays

In a typical play, the moves discussed above will be connected with each other using justification pointers, which indicate which values various calls/returns refer to. Below we analyse the shape of non-empty complete plays in the arenas under discussion.

At the beginning, each such play will contain a segment $q_0\, s\, a_0$, where s consists of 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 $\overset{\curvearrowleft}{q_1 \;\; a_1}$-pair as defining a *thread of play*. Moves made between q_1 and a_1 are then said to occur inside 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* inside the corresponding thread $\overset{\curvearrowleft}{q_1 \;\; a_1}$. 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 $\overset{\curvearrowleft}{q_1 \;\; a_1}$. We give a sample play below, which represents the interaction of the term $\lambda x^{\mathsf{unit}}.\lambda f^{\mathsf{unit}\to\mathsf{unit}}.f x$ with context

$$\mathbf{let}\, g = [\,]\, \mathbf{in}\, \mathbf{let}\, f_1 = g()\, \mathbf{in}\, \mathbf{let}\, f_2 = g()\, \mathbf{in}\, f_1(\lambda x^{\mathsf{unit}}.\mathbf{let}\, f_3 = g()\, \mathbf{in}\, ()).$$


q_0 a_0 q_1 a_1 q_1 a_1 q_2 q_*^1 q_1 a_1 a_*^1 a_2

Due to the well-bracketing condition and the availability of q_*^i moves, each thread (including all of its subthreads) may 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, i.e. switches may occur after a P-move made in the right-hand side part of the arena. 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.

We shall introduce an automata-theoretic model over infinite alphabets, called VPCMA, to capture such scenarios. The preceding play will then 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 make it possible to reconstruct uniquely the underpinning pointer structure in game semantics by relating occurrences of q_2 to the corresponding justifer a_1.

There is one more complication due to the visibility condition. Note that once a_2 is played, it will remove the third $q_1 a_1$ 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*. For this reason, we call the ML fragment $\mathrm{RML}_{\mathrm{VPC}}$, where the subscript is short for VPCMA.

5 VISIBLY PUSHDOWN CLASS MEMORY AUTOMATA (VPCMA)

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) [Björklund and Schwentick 2010] and visibly pushdown automata (VPA) [Alur and Madhusudan 2004]. 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 Σ_{push}, Σ_{pop}, and Σ_{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 5.1 (VPCMA). Let $\Sigma = \Sigma_{\mathsf{push}} + \Sigma_{\mathsf{pop}} + \Sigma_{\mathsf{noop}}$ 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:

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

We now explain the workings of a VPCMA. 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 $a \in \Sigma_{\mathsf{push}}$ the automaton can follow a transition $(q, f(d), a, \gamma, q')$ to configuration $(q', f[d \mapsto q'], S \cdot (d, \gamma))$.
- if $a \in \Sigma_{\mathsf{pop}}$ 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 $a \in \Sigma_{\mathsf{noop}}$ 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) $(q, s, a_{\mathsf{push}}, \gamma, p), (q, s, a_{\mathsf{push}}, \gamma', p') \in \Delta \implies \gamma = \gamma', p = p'$;
(ii) $(q, s, a_{\mathsf{pop}}, \gamma, p), (q, s, a_{\mathsf{pop}}, \gamma, p') \in \Delta \implies p = p'$; and
(iii) $(q, s, a_{\mathsf{noop}}, p), (q, s, a_{\mathsf{noop}}, p') \in \Delta \implies p = p'$.

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) [Cotton-Barratt et al. 2015b], 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 5.2. *Emptiness of VPCMA can be reduced to emptiness of weak VPCMA.*

Proof. The key idea of this reduction is to use push-moves when a new data value is introduced, and the corresponding pop-move can check that the data value is in a good state. For simplicity, we will perform all of these pushes at the beginning of the run, and all of the pops at the very end. Hence, if the word w is recognised by the original VPCMA and it has data values $d_1, \ldots, d_n$ occurring in it, the weak VPCMA we construct will recognise the word $(\downarrow, d_1) \cdots (\downarrow, d_n) \cdot w \cdot (\uparrow, d_n) \cdots (\uparrow, d_1)$, where $\downarrow$ and $\uparrow$ are new push- and pop- letters respectively.

Formally, suppose $\mathcal{A} = \left\langle Q, (\Sigma_{\mathsf{push}}, \Sigma_{\mathsf{pop}}, \Sigma_{\mathsf{noop}}), \Gamma, \Delta, q_0, F_G, F_L \right\rangle$ is a VPCMA. We construct a weak VPCMA

$$\mathcal{B} = \left\langle Q \uplus \{\mathtt{start}, \mathtt{end}\}, (\Sigma_{\mathsf{push}} \uplus \{\downarrow\}, \Sigma_{\mathsf{pop}} \uplus \{\uparrow\}, \Sigma_{\mathsf{noop}}), \Gamma \uplus \{*\}, \Delta', \mathtt{start}, F'_G \right\rangle$$

where Δ' is given as follows:

- From the state $\mathtt{start}$ we have the push-transitions allowing new data values: $(\mathtt{start}, \bot, \downarrow, *, \mathtt{start}) \in \Delta'$ and $(\mathtt{start}, \bot, \downarrow, *, q_0) \in \Delta'$.
- For each transition $t \in \Delta$ we have the transition $t[\bot \mapsto \mathtt{start}] \in \Delta'$.
- For each transition $(q, ..., q') \in \Delta$ where $q' \in F_G$, we have the transition $(q, \ldots, \mathtt{end}) \in \Delta'$.
- Finally, for each $q \in F_L$ we have the pop-transition $(\mathtt{end}, q, \uparrow, *, \mathtt{end})$.

F'_G is simply the set $\{\mathtt{end}\}$, unless $q_0 \in F_G$ in which case $F'_G = \{\mathtt{start}, \mathtt{end}\}$ (to catch cases where $\mathcal{L}(\mathcal{A}) = \{\epsilon\}$). It's straightforward to verify that $\mathcal{L}(\mathcal{B}) = \emptyset$ iff $\mathcal{L}(\mathcal{A}) = \emptyset$. □

Using standard product constructions, in the same way as for weak CMA [Cotton-Barratt et al. 2015b], 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

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

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

Scoping VPCMA (SVPCMA)

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 Section 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 5.3. 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, $V \subset_{\mathsf{fin}} \mathcal{D}$ is the set of visible data values, and $S \in (\mathcal{D} \times \Gamma \times \mathcal{P}_{\mathsf{fin}}(\mathcal{D}))^*$. 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 $f(d) = \bot$ or $d \in V$ the automaton can follow transitions as follows:

- if $a \in \Sigma_{\mathsf{push}}$ 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 $a \in \Sigma_{\mathsf{pop}}$ 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 $a \in \Sigma_{\mathsf{noop}}$ 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 $f(d) \neq \bot$ 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 5.4. *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 5.5. *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 Section 8 we show that VPCMA (SVPCMA) emptiness is recursively equivalent to reachability in extended branching VASS [Jacquemard et al. 2016]. In the next section, we use SVPCMA to represent the game semantics of $\mathrm{RML}_{\mathrm{VPC}}$.

6 COMPILING $\mathrm{RML}_{\mathrm{VPC}}$ TO VPCMA

In this section we prove

Theorem 6.1. *Observational equivalence of* $\mathrm{RML}_{\mathrm{VPC}}$*-terms is reducible to the emptiness problem for VPCMA.*

The result, in conjunction with results of Section 8 will imply the left-to-right implication in Theorem 1.1. To establish Theorem 6.1, we rely on the following crucial lemma.

Lemma 6.2. *For any* $\mathrm{RML}_{\mathrm{VPC}}$*-term* $\Gamma \vdash M$*, there exists a weak deterministic SVPCMA* $\mathcal{A}^M$ *whose language is a faithful representation of* $\mathbf{comp}(\llbracket \Gamma \vdash M \rrbracket)$.

As discussed in Section 5, SVPCMA equivalence can be reduced to VPCMA emptiness, so the Lemma implies Theorem 6.1.

Remark 6.2.1. *One may ask if Lemma 6.2 could be restated in a purely game-semantic form, i.e. whether it implies that a syntax-independent class of strategies (including all strategies corresponding to* $\mathrm{RML}_{\mathrm{VPC}}$*-terms) can be represented via SVPCMA. We do not know the answer to that question. The only characterisation of* $\mathrm{RML}_{\mathrm{VPC}}$ *denotations that we currently have is inductive on the syntax of* $\mathrm{RML}_{\mathrm{VPC}}$.

In any case, note that prearenas corresponding to $\mathrm{RML}_{\mathrm{VPC}}$ *types will also admit strategies that are not recursively enumerable, which definitely have to be ruled out. The fact that the decidability status of basic decision problems for SVPCMA is not settled (and turns out to be equivalent to an open problem in Petri net theory) makes it even harder to pin down the conditions under which one could hope for a semantic version of Lemma 6.2.*

We shall prove the Lemma by induction for terms in canonical form.

Definition 6.3. 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 [Hopkins 2012] 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).

Encoding of plays as words

Next we explain how plays will be encoded as data words. Each move m will be represented by a pair (m, d) where $d \in \mathcal{D}$. As already hinted at the end of Section 3, the data values will be used in such a way that they will help us encode justification pointers.

It must be mentioned that some justification pointers are determined uniquely by the underlying sequence of moves and there is no need to represent them. For example, pointers from answers can be reconstructed via the Well-Bracketing condition. In general, pointers from questions 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 [Hopkins et al. 2011], called the *O-strict fragment*. RML$_{\mathrm{VPC}}$ 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 Figure 8. Consequently, our scheme will only need to account for such pointers.

We give the formal definition below, referring to moves from Figure 8. Here are the main features of the encoding.

(1) Each q_1-move must be accompanied by a fresh data value, i.e. one different from any data value used thus far. This makes the data value suitable as a thread identifier (see Section 4).
(2) Each q_2-move will take the same data value as their justifying a_1-move. This amounts to an unambiguous representation of the pointer from q_2.

In other cases, moves will simply take the same data value as the preceding move. In particular, this will apply to moves corresponding to the types of free variables.

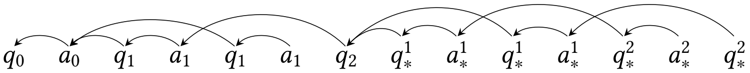
Definition 6.4. Given an arena A corresponding to an RML$_{\mathrm{VPC}}$ typing judgment, we define an encoding relation $\models \subseteq (M_A \times \mathcal{D})^* \times P_A$ as follows ($w \models s$ stands for "data word w is an encoding of play s").

$$\begin{array}{rcll}
(q_0, d) & \models & q_0 & d \in \mathcal{D} \\
w\,(q_1, d) & \models & s q_1 & w \models s \text{ and } d \text{ does not occur in w} \\
w_1\,(a_1, d)\,w_2\,(q_2, d) & \models & s_1\ \overset{\curvearrowleft}{a_1\ s_2\ q_2} & w_1(a_1, d) \models s_1 a_1 \text{ and } w_1(a_1, d) w_2 \models s_1 a_1 s_2 \\
w_1\,(q_*^j, d)\,w_2\,(a_*^j, d) & \models & s_1\ \overset{\curvearrowleft}{q_*^j\ s_2\ a_*^j} & w_1(q_*^j, d) \models s_1 q_*^j \text{ and } w_1(q_*^j, d) w_2 \models s_1 q_j^* s_2 \\
w(m, d)(m', d) & \models & s m m' & w(m, d) \models s m \text{ and } (m' \text{ is a P-move or } m' \neq q_0, q_1, q_2, a_*^j)
\end{array}$$

By our earlier discussion, the encoding scheme is faithful for O-pointers: whenever $w \models s_1$ and $w \models s_2$, we can be sure that s_1 and s_2 consist of the same sequences of moves, and pointers emanating from O-moves will point at (the respective copies of) the same P-moves.

However, it turns out that 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. Specifically, these are the P-moves of the form q^i, q_*^i for $i > 1$ from Figure 8. Fortunately, the same problem was are already present in the O-strict fragment and has been solved using the marking technique from [Hopkins et al. 2011]. The technique represents pointers by marking the source m and target n of each pointer to be represented: technically, this is achieved by creating a copy of the move alphabet and using $\widehat{m}, \widehat{n}$ (instead of m and n respectively) to indicate the pointer. If all pointers were to be represented in this way simultaneously, the representation would not be faithful. Thus, instead, the representation of a single play consists of *several* sequences: one for each problematic pointer (in which only the relevant pointer is represented explicitly) as well as a sequence in which no pointers are represented (the latter facilitates modular construction). Because the pointers are from P-moves and the strategies under consideration are deterministic, the distributed representation of single pointers suffices to represent pointer information faithfully.

Example 6.5. Consider the following play



in prearena int → (int → int → int) → int. The pointer information is then faithfully by the following data words.

$$(q_0,d)(a_0,d)(q_1,d_1)(a_1,d_1)(q_1,d_2)(a_1,d_2)(q_2,d_1)(q^1_*,d_1)(\widehat{a^1_*},d_1)(q^1_*,d_1)(a^1_*,d_1)(\widehat{q^2_*},d_1)(a^2_*,d_1)(q^2_*,d_1)$$
$$(q_0,d)(a_0,d)(q_1,d_1)(a_1,d_1)(q_1,d_2)(a_1,d_2)(q_2,d_1)(q^1_*,d_1)(a^1_*,d_1)(q^1_*,d_1)(\widehat{a^1_*},d_1)(q^2_*,d_1)(a^2_*,d_1)(\widehat{q^2_*},d_1)$$
$$(q_0,d)(a_0,d)(q_1,d_1)(a_1,d_1)(q_1,d_2)(a_1,d_2)(q_2,d_1)(q^1_*,d_1)(a^1_*,d_1)(q^1_*,d_1)(a^1_*,d_1)(q^2_*,d_1)(a^2_*,d_1)(q^2_*,d_1)$$

The prearenas used to model $\mathrm{RML}_{\mathrm{VPC}}$ differ from those used in [Hopkins et al. 2011] only by the extra moves q_1, a_1. Only a_1 is a P-move and, because it is an answer, the associated pointers are not problematic. Thanks to that fact, once we invoke inductively the construction from [Hopkins et al. 2011], we will never have to add any other explicit representations of P-pointers.

SVPCMA constructions

Next we discuss the automata constructions, focussing on the new cases with respect to [Hopkins et al. 2011], 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 [Hopkins et al. 2011], 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 [Hopkins et al. 2011], 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 $\overset{\curvearrowleft}{q_1\ a_1}$ moves of that thread) being visible.

For the construction, we know that there is a family of VPA, $(\mathcal{A}^M_i)$, where $\mathcal{A}^M_i$ 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 $\mathcal{A}_{(\gamma,i_x)}^M$ (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 $\mathcal{A}_{(\gamma,i_x)}^M$ together with the new state (2).
- The transitions are defined as follows:
  - There is a (no-op) transition $(1) \xrightarrow{a_0,\perp} (2)$
  - For each i_x there is a (no-op) transition $(2) \xrightarrow{i_x,\perp} q_{i_x}$ where q_{i_x} is the initial state of $\mathcal{A}_{(\gamma,i_x)}^M$
  - For each $\mathcal{A}_{(\gamma,i_x)}^M$:
    - For each no-op transition $q_1 \xrightarrow{m} q_2$ inside $\mathcal{A}_{(\gamma,i_x)}^M$, 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 $\mathcal{A}_{(\gamma,i_x)}^M$, 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.

As the construction amounts to running multiple copies of a VPA in parallel, in general the resultant word may contain representations of several points from P-moves. This can be restricted to our convention of representing at most one pointer by using the state of the automaton to monitor whether a pointer has already been represented.

The remaining cases concern **let** $x = \ldots$ **in** M and adaptations of the corresponding cases in the O-strict constructions in the O-strict case [Hopkins 2012; Hopkins et al. 2011]. 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.

let x = **ref** 0 **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 i_x-transitions now change the x-component to i, and answers to $\star_x$-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.

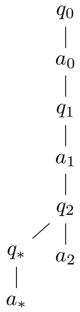


Fig. 9. Shape of prearena for $\vdash \mathsf{int} \to (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$

let $x^\beta = N$ **in** M. $[\![$**let** $x^\beta = N$ **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.

7 REDUCING (S)VPCMA EMPTINESS TO RML$_{\mathrm{VPC}}$ OBSERVATIONAL EQUIVALENCE

So far we have shown that observational equivalence of terms in RML$_{\mathrm{VPC}}$ 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 RML$_{\mathrm{VPC}}$-terms, we will first alter the given SVPCMA to make the reduction to RML-terms easier. We already saw, in Section 5, 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 $\mathcal{A}'$?

The arena that will be used in the argument is given in Figure 9. We shall represent each data value by a single $\overset{\curvearrowleft}{q_1\;\; a_1}$-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 $\overset{\curvearrowleft}{q_2\;\; a_2}$-moves justified by the $\overset{\curvearrowleft}{q_1\;\; a_1}$ 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 $\overset{\curvearrowleft}{q_2\;\; q_*}$-moves, with the stack letter stored locally. If the push-move introduces a fresh data value, the $\overset{\curvearrowleft}{q_1\;\; a_1}$-thread must be created first, and then immediately followed by the $\overset{\curvearrowleft}{q_2\;\; q_*}$-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 with the help of auxiliary $\overset{\curvearrowleft}{q_1 \;\; a_1}$-segments, where the q_1-move provides int-input to determine which transition will be fired next. If the input corresponds to a transition on a fresh data value then the same segment $\overset{\curvearrowleft}{q_1 \;\; a_1}$-segment will also be used to represent the data value, so if the transition is also a noop transition, it will be simulated at the same time.

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

Proposition 7.1. *Given a weak SVPCMA such that on arrival at a final state the stack is always empty and a noop transition is executable, there are* $\mathrm{RML}_{\mathrm{VPC}}$*-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.

Proof. Suppose we have a weak SVPCMA $(Q, \Sigma, \Gamma, \Delta, q_0, F_G)$ where, as previously described, the automaton can only arrive at a final state with an empty stack. We note two ways the transition relation Δ can be partitioned. We will use Δ_{push}, Δ_{pop}, and Δ_{noop} to refer to the classes of push, pop, and noop moves respectively. We will also use the sets Δ_{fresh} and Δ_{old}, consisting of the sets of moves taking a fresh data value, or not, respectively. We shall use injective functions to match the states and transitions of the SVPCMA to the int-type values. In particular, we will have injective functions: $[\![-]\!] : Q \uplus \{\bot\} \to \mathbb{N}$, $[\![-]\!] : \Delta \uplus \{Unknown\} \to \mathbb{N}$; and $[\![-]\!] : \Gamma \to \mathbb{N}$.

For technical convenience, to each state in F_G, we shall add a special noop transition on fresh data values, which will help us recognise arrival at final states. Note that such transitions can always fire as soon as the state is reached. Accordingly, the terms M, N will be taken to be the same as the term shown in Fig. 10 except that the assertion in line 9 (**assert**(! State $\notin [\![F_G]\!]$)) will be omitted in N. We now discuss how the term simulates an SVPCMA. It begins by setting up two global variables.

- The State variable will keep track of the state the automaton is in, taking values from $[\![Q]\!]$.
- Next_Move is used to keep track of the which $\delta \in \Delta$ is to be followed next, taking values from $[\![\Delta]\!]$ to represent this. At the start of the run, or once a transition has been simulated, the variable is reset to a special value $[\![Unknown]\!]$ until O specifies which transition is to be played next.

Following the setup of these variables, the first λ-abstraction occurs in the term. The following part of the term, then, corresponds to how P plays inside $\overset{\curvearrowleft}{q_1 \;\; a_1}$-moves. In particular, the assertions make sure that a_1 can be played only if the next move is unspecified and the input x corresponds to a genuine transition. If this is the case, Next_Move is updated to x and additional actions are performed if the selected transition uses a fresh data value. Note that such transitions are either noop or push transitions, because in SVPCMA the name used in a pop transition must match the one used in the corresponding push transition.

- For noop transitions, the whole transition is simulated (by updates to State and the class memory function CMF) and Next_Move can then be reset to $[\![Unknown]\!]$.
- For push transitions, the local variable CMF is then initialised to $\bot$, but Next_Move is not reset as the simulation is not complete yet (the completion $\overset{\curvearrowleft}{q_2 \;\; q_*}$ is handled by a different part of the term).

```
let
  State = ref $[\![q_0]\!]$
  Next_Move = ref $[\![Unknown]\!]$
in
  $\lambda x^{\mathsf{int}}$.
    let
      CMF = ref –
    in
      assert (!State $\notin [\![F_G]\!]$);
      assert (!Next_Move = $[\![Unknown]\!]$); assert (x $\in [\![\Delta]\!]$);
      Next_Move := $x$;
        if (!Next_Move $\in [\![\Delta_{push} \cap \Delta_{fresh}]\!]$) then CMF := $[\![\bot]\!]$
        else if (!Next_Move $\in [\![\Delta_{noop} \cap \Delta_{fresh}]\!]$) then
                 ($\bigcup_{(q,\bot,\sigma,q')\in\Delta_{noop}\cap\Delta_{fresh}}$ :
                      if (!Next_Move = $[\![(q,\bot,\sigma,q')]\!]$) then
                      (assert (!State = $[\![q]\!]$);
                      State := $[\![q']\!]$; CMF := $q'$;
                      Next_Move := $[\![Unknown]\!]$));
      $\lambda f^{\mathsf{unit}\to\mathsf{unit}}$.
        assert (!Next_Move $\neq [\![Unknown]\!]$);
        if (!Next_Move $\in [\![\Delta_{noop} \cap \Delta_{old}]\!]$) then
          ($\bigcup_{(q,s,\sigma,q')\in\Delta_{noop}\cap\Delta_{old}}$ :
            if (!Next_Move = $[\![(q,s,\sigma,q')]\!]$) then
              (assert (!State = $[\![q]\!]$); assert (!CMF = $[\![s]\!]$);
              State := $[\![q']\!]$; CMF := $q'$;
              Next_Move := $[\![Unknown]\!]$))
        else if (!Next_Move $\in [\![\Delta_{push}]\!]$) then
          let
            Stack_Letter = ref –
          in
            ($\bigcup_{(q,s,\sigma,\gamma,q')\in\Delta_{push}}$ :
              if (!Next_Move = $[\![(q,s,\sigma,\gamma,q')]\!]$) then
                (assert (!State = $[\![q]\!]$); assert (!CMF = $[\![s]\!]$);
                State := $[\![q']\!]$; CMF := $q'$; Stack_Letter := $[\![\gamma]\!]$;
                Next_Move := $[\![Unknown]\!]$));
            f ();
            assert (!Next_Move $\in [\![\Delta_{pop}]\!]$);
            ($\bigcup_{(q,s,\sigma,\gamma,q')\in\Delta_{pop}}$ :
              if (!Next_Move = $[\![(q,s,\sigma,\gamma,q')]\!]$) then
                (assert (!State = $[\![q]\!]$); assert (!CMF = $[\![s]\!]$);
                assert (!Stack_Letter = $[\![\gamma]\!]$);
                State := $[\![q']\!]$; CMF := $q'$;
                Next_Move := $[\![Unknown]\!]$))
        else assert False;
```

Fig. 10. The term encoding SVPCMA emptiness

In both cases above the $\overset{\curvearrowleft}{q_1 \; a_1}$-moves become representatives of the fresh data value and further transitions using that value are simulated by referring to that segment. If the selected transition is not fresh then the segment merely represents the choice of transition by O and will not be revisited (CMF remains set to -).

The second λ-abstraction level corresponds to O making a q_2-move. Because of the following assertion, q_2 can trigger a response only if the next move has already been specified by O. There are two possibilities here.

- If a noop-move is to be made that reads an existing data value. In this case, the first branch of the **if** is taken, and the changes to the existing variables are made as expected. In particular, assertions check that the class memory value for the corresponding $\overset{\curvearrowleft}{q_1 \; a_1}$ segment is indeed compatible with the transition function, which guarantees that only $\overset{\curvearrowleft}{q_1 \; a_1}$-threads corresponding to data values (as opposed to those representing exclusively move choices) can be used to support such moves.

- If a push-move is being made (and so P must follow the q_2-move up with a q_*), the second branch of the **if** is taken, and the local variable Stack_Letter is set up – this variable will store the letter of Γ that is pushed to the stack with this push. In this branch, the term will evaluate f **()**, which corresponds to the q_* move. Before playing this, the variables State, CMF, and Stack_Letter are all adjusted as per the transition that is being made. Also, Next_Move is set to $[\![Unknown]\!]$, so that after the q_* move O will be able to play a q_1-move specifying the next transition to take.

The code following f **()** handles the situation when O plays the move a_*, which is used to simulate pops. Note that the fact that at this point the scope of the variable CMF is still that of the $\overset{\curvearrowleft}{q_1 \;\; a_1}$-thread that made the corresponding push-move, and so the same data value that was used in the push move will be used in the pop-move. To determine whether P will respond to a_* we first check if a pop transition is indeed to be simulated and whether the current stack content, class memory and state are consistent with it.

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 available noop transition. Consequently, final-state reachability can be checked through equality on complete plays generated by M and N respectively.

□

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 $\overset{\curvearrowleft}{q_1 \;\; a_1}$ segments as there are transitions and O-choice could be represented by playing a $\overset{\curvearrowleft}{q_2 \;\; a_2}$ 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 unit → (unit → unit) → unit.

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

8 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 [Jacquemard et al. 2016] 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) [de Groote et al. 2004], 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 8.1 (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:

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

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:

$$\text{(unary)} \quad \frac{q_1, \bar{v} + \bar{v}_1}{q_2, \bar{v} + \bar{v}_2} \qquad\qquad \text{(split)} \quad \frac{q_1, \bar{v} + \bar{v}' + \bar{v}_1}{q_2, \bar{v} + \bar{v}_2 \qquad q_3, \bar{v}' + \bar{v}_3}$$

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 to restrict unary transitions to only a single increment or decrement, and require split transitions to make no increments or decrements. It is clear that our 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 [Jacquemard et al. 2016], and extend BVASS with the ability to split counters in more complex ways when a split transition is made.

Definition 8.2 (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_1, i_2, i_3) can fire *any* number of times when a split transition is made, and for each time it fires it will decrement the i_1th counter (pre-splitting), and then increment the i_2th counter in the left-hand branch and the i_3th 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 $\bar{v}_2, \bar{v}_3 \in \mathbb{N}^k$ such that $\bar{v} = \bar{u}_1 + \bar{v}_2 + \bar{v}_3 + \Sigma(n_i \cdot \bar{e}_{\pi_1(c_i)})$, and the left child node is labelled $(q', \bar{u}_2 + \bar{v}_2 + \Sigma(n_i \cdot \bar{e}_{\pi_2(c_i)}))$, and the right child is labelled $(q'', \bar{u}_3 + \bar{v}_3 + \Sigma(n_i \cdot \bar{e}_{\pi_3(c_i)}))$, 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 $q \in L$. The *reachability problem* asks whether there is an accepting run with root node labelled $(q_0, \bar{0})$.

Remark 8.2.1. 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 [Jacquemard et al. 2016, Sec. 5], our definition of runs corresponds to the non-commutative treatment of constraints.

8.1 From VPCMA to EBVASS

Theorem 8.3. *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. (Theorem 5.2) we work with weak VPCMA. Suppose $\mathcal{A} = \left\langle Q, \Sigma, \Gamma, \Delta, q_0, \{q_f\} \right\rangle$ 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 $a \in \Sigma_{\mathsf{noop}}$, we have:

$$\textsc{(no-op)} \quad \frac{(q, p), \bar{v} + \bar{e}_{s,s'}}{(q', p), \bar{v} + \bar{e}_{q',s'}}$$

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

$$\textsc{(push-pop)} \quad \frac{(q_1, p), \bar{v}_1 + \bar{v}_2 + \bar{e}_{s,s''}}{(q_2, q_3), \bar{v}_1 + \bar{e}_{q_2,s'} \qquad (q_4, p), \bar{v}_2 + \bar{e}_{q_4,s''}}$$

(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.)

[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.

- For every $x \in Q \times Q$ and $q \in Q$ we have the rule

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

(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 $q \in Q$:

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

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

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

First we note a simple property of VPCMA:

Lemma 8.4. *If $\mathcal{A}$ is a VPCMA and $(q_1, f_1, \epsilon) \twoheadrightarrow_{\mathcal{A}} (q_2, f_2, \epsilon)$ then for any valid stack S $(q_1, f_1, S) \twoheadrightarrow_{\mathcal{A}} (q_2, f_2, S)$.*

Where $\twoheadrightarrow_{\mathcal{A}}$ is the reflexive transitive closure of the transition relation on configurations of $\mathcal{A}$.

Proof. This is a straightforward copying of the run to the case where a stack is already present. □

We further make the following definition: where $\bar{v}$ is a positive vector over pairs of states (i.e. formed from the vectors $e_{q,q'}$ defined above), and f_1, f_2 are class memory functions over a data set $\mathcal{D}$, we say $\bar{v} \sim (f_1, f_2)$ just if the following is true: For each $e_{q,q'}$ in the unit-vector decomposition of $\bar{v}$ there is a data value $d \in \mathcal{D}$ such that $f_1(d) = q$ and $f_2(d) = q'$. Further, all other data values d' are such that $f_1(d') = \perp = f_2(d')$.

We show that the meanings we ascribed to the counters and states of the EBVASS match the formal properties of the system, in the following lemma.

Lemma 8.5. *Given a VPCMA $\mathcal{A}$ and the corresponding constructed EBVASS $\mathcal{E}_{\mathcal{A}}$, there is the following correspondence:*

If there is an accepting run of $\mathcal{E}_{\mathcal{A}}$ starting at a configuration $((q_1, q_2), \bar{v})$ then: for all class memory functions f_1, f_2 such that $\bar{v} \sim (f_1, f_2)$ there is a class memory function f_2' extending f_2 such that $(q_1, f_1, \epsilon) \twoheadrightarrow_{\mathcal{A}} (q_2, f_2', \epsilon)$.

Note that for f' to extend f we mean that for all d such that $f(d) \neq \perp$, $f'(d) = f(d)$.

Proof. We prove this by induction on the construction of the EBVASS run.

Base Case. The smallest possible runs are single-node trees, so the root is also a leaf. This means the configuration is of the form $((q, q), \bar{0})$. In this case the VPCMA run of the same is the trivial run.

Inductive Step. We deal with the various possible initial steps for the EBVASS run.

increment. Suppose there is a run of $\mathcal{E}_{\mathcal{A}}$ with root configuration $((q_1, q_2), \bar{v})$ such that the first step of the run is an increment-transition. Hence there is a a run of $\mathcal{E}_{\mathcal{A}}$ starting with a root configuration $((q_1, q_2), \bar{v} + e_{\perp,q})$. Let f_1, f_2 be class memory functions such that $\bar{v} \sim (f_1, f_2)$. Let $d \in \mathcal{D}$ be such that $f_1(d) = \perp = f_2(d)$. Then $\bar{v} + e_{\perp,q} \sim (f_1, f_2[d \mapsto q])$ so by the inductive hypothesis there is an f_2' extending $f_2[d \mapsto q]$ such that $(q_1, f_1, \epsilon) \twoheadrightarrow_{\mathcal{A}} (q_2, f_2', \epsilon)$. Note that by construction this f_2' also extends f_2, and we are done.

DECREMENT. Suppose there is a run of $\mathcal{E}_{\mathcal{A}}$ with root configuration $((q_1, q_2), \bar{v} + e_{q,q})$ such that the first step of the run is a DECREMENT-transition (of $e_{q,q}$). Let f_1, f_2 be class memory functions such that $\bar{v} + e_{q,q} \sim (f_1, f_2)$, with data value d such that $f_1(d) = q = f_2(d)$. Note that in this case $\bar{v} \sim (f_1[d \mapsto \bot], f_2[d \mapsto \bot])$, so by inductive hypothesis there is some f_3 extending $f_2[d \mapsto \bot]$ such that $(q_1, f_1[d \mapsto \bot], \epsilon) \twoheadrightarrow_{\mathcal{A}} (q_2, f_3, \epsilon)$. WLOG assume $f_3(d) = \bot$ (since all data values mapped to $\bot$ are interchangeable, this can be assumed). Then $f_3[d \mapsto q]$ extends f_2, and it is straightforward to check that the same transitions show $(q_1, f_1, \epsilon) \twoheadrightarrow_{\mathcal{A}} (q_2, f_3[d \mapsto q], \epsilon)$.

NO-OP. Suppose there is a run of $\mathcal{E}_{\mathcal{A}}$ with root configuration $((q_1, q_2), \bar{v} + e_{s,s'})$ such that the first step of the run is a NO-OP-transition generated by a transition $(q_1, s, a, q_1') \in \Delta$ (so the configuration after the first step is $((q_1', q_2), \bar{v} + e_{q_1',s'})$). Given f_1, f_2 s.t. $\bar{v} + e_{s,s'} \sim (f_1, f_2)$, let $f_1' = f_1[d \mapsto q_1']$ where d is s.t. $f_1(d) = s$ and $f_2(d) = s'$. Then $\bar{v} + e_{q_1',s'} \sim (f_1', f_2)$ so by IH there is an f_2' extending f_2 s.t. $(q_1', f_1', \epsilon) \twoheadrightarrow_{\mathcal{A}} (q_2, f_2', \epsilon)$. It is a straightforward check that by following the transition $(q_1, s, a, q_1') \in \Delta$, $(q_1, f_1, \epsilon) \rightarrow_{\mathcal{A}} (q_1', f_1', \epsilon)$ and so by the construction of $\twoheadrightarrow_{\mathcal{A}}$ we are done.

PUSH-POP. Suppose there is a run of $\mathcal{E}_{\mathcal{A}}$ with root configuration $((q_1, p), \bar{v}_1 + \bar{v}_2 + e_{s,s''})$ with first step of the run a PUSH-POP-transition generated by push-transition $(q_1, s, a, \gamma, q_2) \in \Delta$ and pop-transition $(q_3, s', b, \gamma, q_4) \in \Delta$. This is the most complicated step. The central idea is that the left-hand branch of the EBVASS will simulate the run between the push and pop moves, and the right-hand branch will simulate the rest of the run. Hence, the shape of our argument will be that given appropriate f_1 and f_p, there exist f_2, f_3', f_4, f_p' such that:

$$(q_1, f_1, \epsilon) \rightarrow_{\mathcal{A}} (q_2, f_2, \gamma) \twoheadrightarrow_{\mathcal{A}} (q_3, f_3', \gamma) \rightarrow_{\mathcal{A}} (q_4, f_4, \epsilon) \twoheadrightarrow_{\mathcal{A}} (p, f_p', \epsilon)$$

The constraints allow data values to be "moved" in the left-hand branch such that they are then in different starting positions for the rest of the run. Hence, let d be a data value such that $f_1(d) = s$ and $f_p(d) = s''$. Then let $f_2 = f_1[d \mapsto q_2]$, and the first step of the above is shown.

We now construct f_3 using details on exactly how the constraints fired. For simplicity, we assume every counter increment was "split" by the constraints firing – it is straightforward to adjust the below to allow otherwise. Suppose we had k-firings of the constraints, labelled $c_1, \ldots, c_k$, where c_i led to splitting an increment $e_{q_i,q_i''}$ into $e_{q_i,q_i'}$ in the left branch and $e_{q_i',q_i''}$ in the right branch. Then choose data values $d_1, \ldots, d_k$ (distinct from d) such that $f_1(d_i) = q_i$ and $f_p(d_i) = q_i''$ (which exist by assumptions on f_1 and f_p). We then let $f_3 = f_2[d_i \mapsto q_i', d \mapsto s']$. Then, if $\bar{u}$ is the counter vector of the left-hand branch after the split, by construction we have $\bar{u} \sim (f_2, f_3)$, so by IH there is some f_3' extending f_3 with $(q_2, f_2, \epsilon) \twoheadrightarrow_{\mathcal{A}} (q_3, f_3', \epsilon)$. By Theorem 8.4 we get $(q_2, f_2, \gamma) \twoheadrightarrow_{\mathcal{A}} (q_3, f_3', \gamma)$.

f_4 is then simply $f_3'[d \mapsto q_4]$, and it is by following the chosen pop-transition that we see $(q_3, f_3', \gamma) \rightarrow_{\mathcal{A}} (q_4, f_4, \epsilon)$. Now suppose $\bar{u'}$ is the counter vector of the right-hand branch after the split. It is a simple argument that f_4 is extending some class memory function f_4' such that $\bar{u'} \sim (f_4', f_p)$. Then by IH there is some f_p'' such that $(q_4, f_4', \epsilon) \twoheadrightarrow_{\mathcal{A}} (p, f_p'', \epsilon)$. Now we can choose the extensions of f_p'' over f_p not to clash with the extensions of f_4 over f_4', and we define f_p' to be f_p'' augmented with the extensions f_4 has over f_4'. Then it is clear that $(q_4, f_4, \epsilon) \twoheadrightarrow_{\mathcal{A}} (p, f_p', \epsilon)$. □

From the above we get the first half of Theorem 8.3: if there is an accepting run of $\mathcal{E}_{\mathcal{A}}$ from $((q_0, q_f), \bar{0})$ then this lemma gives us that there is an accepting run of $\mathcal{A}$. We now show the converse. We say that for class memory functions f and f', f' *is reachable from* f iff $f(d) \neq \bot$ implies $f'(d) \neq \bot$. For pairs of class memory functions (f, f') such that f' is reachable from f we define the corresponding counter vector, $\bar{v}_{f,f'}$ as follows: the value in the counter corresponding to the pair $(s, s') \in Q_{\bot} \times Q$ is equal to the number of data values d such that $f(d) = s$ and $f'(d) = s'$.

Lemma 8.6. *Suppose there is a run of $\mathcal{A}$, $(q_1, f_1, \epsilon) \twoheadrightarrow_{\mathcal{A}} (q_2, f_2, \epsilon)$. Then for any $q_3 \in Q$ and f_3 reachable from f_2, there is a run of $\mathcal{E}_{\mathcal{A}}$ with root $((q_1, q_3), \bar{v}_{f_1,f_3})$ such that the rightmost leaf node is $((q_2, q_3), \bar{v}_{f_2,f_3})$, and all other leaves are labelled with the* 0*-vector and leaf states.*

Proof. We prove this by induction on the construction of the run.

Base Case. Here we consider case where the run is of length 1. In this case we must have the only transition taken is a transition $(q_1, a, s, q_2) \in \Delta$ where $a \in \Sigma_{\mathsf{noop}}$, and $f_2 = f_1[d \mapsto q_2]$ where $f_1(d) = s$. From the root node $((q_1, q_3), \bar{v}_{f_1,f_3})$ $\mathcal{E}_{\mathcal{A}}$ can follow the no-op rule corresponding to the transition arriving at configuration $((q_2, q_3), \bar{v}_{f_2,f_3})$, and this is then the only (so rightmost) leaf.

Inductive Step. We assume the run is of length ≥ 2. In this case either there is a non-trivial decomposition $(q_1, f_1, \epsilon) \twoheadrightarrow_{\mathcal{A}} (q', f', \epsilon) \twoheadrightarrow (q_2, f_2, \epsilon)$ or there is not. We consider these two cases in turn.

- Suppose there is such a decomposition, and let q_3 and f_3 be given. Note that clearly f' is reachable from f_1. Then by IH there is a run of $\mathcal{E}_{\mathcal{A}}$ starting at $((q_1, q_3), \bar{v}_{f_1,f_3})$ with rightmost leaf node $((q', q_3), \bar{v}_{f',f_3})$. Similarly there is a run starting at $((q', q_3), \bar{v}_{f',f_3})$ with rightmost leaf $((q_2, q_3), \bar{v}_{f_2,f_3})$. By composing these two runs in the obvious fashion, we obtain the required run.
- Suppose there is no such decomposition. Then the run begins with a push-move that is only popped as the last move of the run, and we can decompose the run as follows:

$$(q_1, f_1, \epsilon) \rightarrow_{\mathcal{A}} (q', f', (d_0, \gamma)) \twoheadrightarrow_{\mathcal{A}} (q'', f'', (d_0, \gamma)) \rightarrow_{\mathcal{A}} (q_2, f_2, \epsilon)$$

  where $f' = f_1[d_0 \mapsto q']$ and $f_2 = f''[d_0 \mapsto q_2]$. Now it is a straightforward check that, by using the push-pop-rule corresponding to the push and pop transitions used in the run of $\mathcal{A}$ together with appropriate firing of constraints, $\mathcal{E}_{\mathcal{A}}$ can split from state $((q_1, q_3), \bar{v}_{f_1,f_3})$ into $((q', q''), \bar{v}_{f',f''})$ (on the left) and $((q_2, q_3), \bar{v}_{f_2,f_3}$ on the right. Hence, it just remains to show that the left subtree of this has a run resulting only in accepting leaves.

  Now, since by assumption the stack symbol γ isn't removed from the stack in the run from $(q', f', (d_0, \gamma))$ to $(q'', f'', (d_0, \gamma))$, we also have that $(q', f', \epsilon) \twoheadrightarrow_{\mathcal{A}} (q'', f'', \epsilon)$. So by the IH there is a run of $\mathcal{E}_{\mathcal{A}}$ starting at $((q', q''), \bar{v}_{f',f''})$ with rightmost leaf $((q'', q''), \bar{v}_{f'',f''})$. It is trivial to see that repeated application of decrement rules will be able to reduce this rightmost leaf to an accepting leaf.

□

Hence, if there is an accepting run of $\mathcal{A}$, it takes the form $(q_0, f_0, \epsilon) \twoheadrightarrow_{\mathcal{A}} (q_f, f', \epsilon)$, and so there is a run from $((q_0, q_f), \bar{v}_{f_0,f'})$ with rightmost leaf $((q_f, q_f), \bar{v}_{f',f'})$. Now by repeated application of the decrement-rule, this rightmost leaf can be reduced to an accepting leaf. Furthermore, it's straightforward to see that starting from $((q_0, q_f), \bar{0})$ repeated application of the increment-rule yields a route to $((q_0, q_f), \bar{v}_{f_0,f'})$. By putting these partial-runs together, we obtain the result that if there is an accepting run of $\mathcal{A}$ then there is an accepting run of $\mathcal{E}_{\mathcal{A}}$, completing the proof of Theorem 8.3.

8.2 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 5.5) we do this by reducing EBVASS reachability to SVPCMA emptiness. The key idea is that words over a visibly pushdown alphabet can be viewed as their construction trees when generated by the grammar:

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

Thus 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. To represent splits we will use tags $\mathsf{split}_{\mathsf{push}}$ and $\mathsf{split}_{\mathsf{pop}}$.

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. We will be able to fire transitions corresponding to the constraints before the push-move of a split-transition. Given a constraint (i, j, k) the corresponding transition will take a data value where the class memory function remembers it as belonging to counter i, that value will be deactivated and a new value will be created to represent (j, k): first j and then k.

Remark 8.6.1. *(1) Class memory functions are normally of the form $f : \mathcal{D} \longrightarrow Q_\perp$. In our encoding we shall use a special set* Lab *of* labels *to keep track of local behaviour and will rely on functions $f : \mathcal{D} \longrightarrow \mathsf{Lab}_\perp$ instead. Accordingly, our SVPCMA will have a transition relation of the form*

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

Note that the above can easily be 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 convenient when reducing VPCMA to EBVASS, we will now find it useful to restrict unary transitions to a single increment or a single decrement, and to require split transitions to make no increments or decrements.

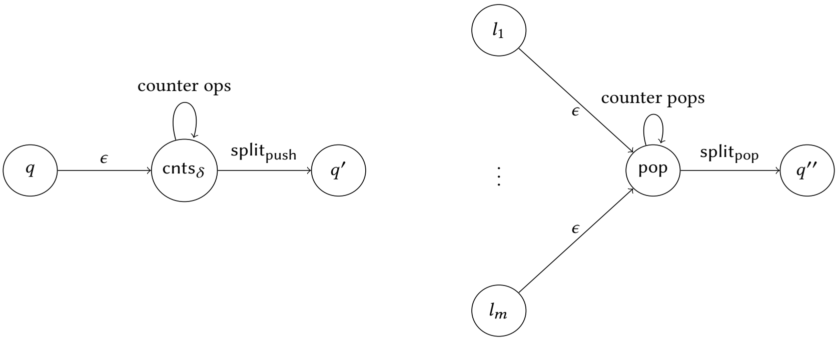
In our reduction, data values will be used to store the counter information. The value of a counter can be represented by the number of data values which the class memory function assigns a label corresponding to that counter (we use the labels $1, \ldots, n$ for the n counters). When a counter is incremented, 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, in addition to the basic intuition explained above, we will also rely on a slightly more complicated scheme: a constraint (i, j, k) will be interpreted in the same way as decrementing i and creating a fresh data value, initially labelled with (j, k). The label reflects the fact that the value represents j at the start and, after it is used up as j, it will represent k. Using the stack, we will make sure that the decrement related to k never occurs under the decrement related to j, where “under” refers to the tree traced out by the automaton. Consequently, there will exist a split node such that the two uses occur respectively in the left and right subtrees of that node. To prevent k from being used prematurely, the decrement as j will be accompanied by a push where k is put on the stack and the label is set to done until a matching pop move resets it to k. Note that the matching pop ensures that all nodes under the decrement related to j have already been visited.

We fire transitions corresponding to constraints immediately before interpreting a split. This creates new data values whose scope extends over both left and right subtrees generated from the split. Note, though, that the encoding will not enforce the fact that the newly created labels (j, k) are used up in the left and right subtrees resulting from the split. Rather, the exact point where the constraint fires in the EBVASS is determined implicitly as the least common ancestor of the nodes corresponding to decrementing the j and k counters respectively.

Because of the presence of labels (j, k), there will be two kinds of decrements. Those using numerical labels will simply reset the class to done without affecting the stack, whereas those on label (j, k) will induce a push move (k will be pushed). The two kinds will be tagged with $\mathsf{dec}_{\mathsf{noop}}$ and $\mathsf{dec}_{\mathsf{push}}$ respectively. In contrast, increments will always be tagged with $\mathsf{inc}_{\mathsf{noop}}$.

Altogether, the shape of the parts of the automaton corresponding to a split transition $\delta = (q, q', q'')$ is shown below, where we illustrate the transitions with tags.



- "Counter ops" stands for a collection of loops, one for each constraint, which enable the interpretation of a constraint. Given $c = (i, j, k) \in C$, the loop will have the shape $\mathsf{cnts}_\delta \xrightarrow{\mathsf{dec}_{\mathsf{noop}}/\mathsf{dec}_{\mathsf{push}}} c_\delta \xrightarrow{\mathsf{inc}_{\mathsf{noop}}} \mathsf{cnts}_\delta$, which will consume i (depending on the kind of the data value) and produce a data value labelled with (j, k).
- "Counter pops" are of the form $\mathsf{pop} \xrightarrow{\mathsf{dec}_{\mathsf{pop}}} \mathsf{pop}$, they correspond to the moment when a data value initially labelled (j, k) starts representing counter k.

Theorem 8.7. *The reachability problem for EBVASS is reducible to the emptiness problem for SVPCMA.*

Proof. Following the discussion above, given an EBVASS $\mathcal{B} = (Q, q_0, L, n, \Delta_u, \Delta_s, C)$, the corresponding SVPCMA $\mathcal{A}_\mathcal{B}$ is defined as follows. The set of states of $\mathcal{A}_\mathcal{B}$ is $Q \uplus \{\mathsf{cnts}_\delta, c_\delta\}_{\delta \in \Delta_s} \uplus \{\mathsf{pop}\}$, where q_0 is initial;

$$\begin{aligned} \mathsf{Lab} &= \{1, \ldots, n\} \cup (\{1, \cdots, n\} \times \{1, \cdots, n\}) \cup \{\mathsf{done}, \mathsf{split}\} \\ \Gamma &= Q + \{1, \ldots, n\} \\ \Sigma &= \{\mathsf{dec}_{\mathsf{push}}, \mathsf{split}_{\mathsf{push}}\} + \{\mathsf{dec}_{\mathsf{pop}}, \mathsf{split}_{\mathsf{pop}}\} + \{\mathsf{inc}_{\mathsf{noop}}, \mathsf{dec}_{\mathsf{noop}}, \epsilon\}. \end{aligned}$$

The set of globally accepting states is L; and the (singleton) set of locally accepting label is $\{\mathsf{done}\}$. The transition relation of $\mathcal{A}_\mathcal{B}$

$$\Delta \subseteq \big(Q \times \mathsf{Lab}_\perp \times (\Sigma_{\mathsf{push}} \cup \Sigma_{\mathsf{pop}}) \times \Gamma \times Q \times \mathsf{Lab}\big) \cup \big(Q \times \mathsf{Lab}_\perp \times \Sigma_{\mathsf{noop}} \times Q \times \mathsf{Lab}\big)$$

is constructed as follows:

(i) for each (unary) increment transition $\delta \in \Delta_u$ of the form $q \xrightarrow{+e_i} q'$ we have the transition $(q, \perp, \mathsf{inc}_{\mathsf{noop}}, q', i)$;

(ii) for each (unary) decrement transition $\delta \in \Delta_u$ of the form $q \xrightarrow{-e_i} q'$ we have the transitions:
- $(q, i, \mathsf{dec}_{\mathsf{noop}}, q', \mathsf{done})$, and
- $(q, (i, x), \mathsf{dec}_{\mathsf{push}}, x, q', \mathsf{done})$ for each $x \in \{1, \ldots, n\}$;

(iii) for each split transition $\delta \in \Delta_s$ of the form $q \longrightarrow q' + q''$ we have

- ϵ-transition $q \xrightarrow{\epsilon} \mathsf{cnts}_\delta$, and
- push-transition $(\mathsf{cnts}_\delta, \bot, \mathsf{split}_{\mathsf{push}}, q'', q', \mathsf{split})$;

(iv) for each $\delta \in \Delta_s$ and constraint $c = (i, j, k) \in C$, we have:

- noop-transition $(\mathsf{cnts}_\delta, i, \mathsf{dec}_{\mathsf{noop}}, c_\delta, \mathsf{done})$,
- push-transitions $(\mathsf{cnts}_\delta, (i, x), \mathsf{dec}_{\mathsf{push}}, x, c_\delta, \mathsf{done})$ for each $x \in \{1, \ldots, n\}$, and
- noop-transition $(c_\delta, \bot, \mathsf{inc}_{\mathsf{noop}}, \mathsf{cnts}_\delta, (j, k))$;

(v) for each $i \in \{1, \ldots, n\}$ we have the pop-transition $(\mathsf{pop}, \mathsf{done}, \mathsf{dec}_{\mathsf{pop}}, i, \mathsf{pop}, i)$;

(vi) for each $l \in L$ we have the ϵ-transition $l \xrightarrow{\epsilon} \mathsf{pop}$;

(vii) finally, for each $q \in Q$ we have the pop-transition $(\mathsf{pop}, \mathsf{split}, \mathsf{split}_{\mathsf{pop}}, q, q, \mathsf{done})$.

In the above, we have used "ϵ-transitions" $q \xrightarrow{\epsilon} q'$ on the understanding that they can be readily coded as a noop-transition $(q, \bot, \epsilon, q', \mathsf{done})$.

In case (iv) above, the first two types of transitions represent the two options that may arise when firing the constraint (i, j, k). In order to execute the constraint we need to find "one unit of counter i" and decrement it. That unit could come from

1. a simple increment (in which case the decrement will be modelled by $\mathsf{dec}_{\mathsf{noop}}$), or
2. a constraint (in which case the decrement will be modelled by $\mathsf{dec}_{\mathsf{push}}$).

(I.e. they are the two possible transitions from cnts_δ to c_δ in "counter ops".) In the second case, the data values of interest are labelled with (i, x), where x does not matter, because we only need to know they currently represent "counter i". The $\mathsf{inc}_{\mathsf{noop}}$ transition then ensures that the outcome of the constraint is labelled with (j, k).

Note that, due to the scoping constaint, whenever a fresh name is created (via $\mathsf{inc}_{\mathsf{noop}}$) and ρ is the top of the stack, the scope of that name will end as soon as ρ is popped from the stack. In particular, this means that $\mathsf{split}_{\mathsf{push}}$ will protect fresh values created inside the left subtree after a split from being used in the right subtree, because their scope will be terminated after the corresponding $\mathsf{split}_{\mathsf{pop}}$.

It is easy to see that an accepting run of $\mathcal{B}$ will give rise an accepting run of $\mathcal{A}_{\mathcal{B}}$.

For the converse, we analyse the features of our translation that deviate from the behaviour of EBVASS.

First, in EBVASS, all constraints are fired simultaneously at a split transition, but our simulation handles them sequentially. Thus, we need to show that our interpretation does not lead to interference, whereby a fresh data value introduced via $\mathsf{inc}_{\mathsf{noop}}$ would be consumed by $\mathsf{dec}_{\mathsf{push}}$ during a "Counter ops" loop for the *same* split. Although such a scenario is possible in $\mathcal{A}_{\mathcal{B}}$, it will never develop into an accepting run. Indeed, consider the following transition sequence

$$\cdots c_\delta \xrightarrow{\mathsf{inc}_{\mathsf{noop}}} \mathsf{cnts}_\delta \xrightarrow{\cdots} \mathsf{cnts}_\delta \xrightarrow{\mathsf{dec}_{\mathsf{push}}} c_\delta \xrightarrow{\cdots} \mathsf{cnts}_\delta \xrightarrow{\mathsf{split}_{\mathsf{push}}} q' \cdots l \xrightarrow{\epsilon} \mathsf{pop} \xrightarrow{{\mathsf{dec}_{\mathsf{pop}}}^*} \mathsf{pop} \xrightarrow{\mathsf{split}_{\mathsf{pop}}} q''$$

where $\mathsf{inc}_{\mathsf{noop}}$ creates a fresh data value for (j, k), $\mathsf{dec}_{\mathsf{push}}$ then decrements the same data value, and there is no intervening $\mathsf{split}_{\mathsf{push}}$ between them. Suppose the label of the data value is changed to k by $\mathsf{dec}_{\mathsf{pop}}$ as shown. Now, in order for the data value to evolve further into done, the automaton would have to perform a $\mathsf{split}_{\mathsf{pop}}$-labelled transition (possibly after some more $\mathsf{dec}_{\mathsf{pop}}$) that matches the $\mathsf{split}_{\mathsf{push}}$ of some split δ' that happens *earlier* than the split δ displayed above. However, making the $\mathsf{split}_{\mathsf{pop}}$ transition will then terminate the scope of the data value, so its class will never reach done.

Thus, in accepting runs of $\mathcal{A}_{\mathcal{B}}$, $\mathsf{dec}_{\mathsf{push}}$-labelled transitions must rely on data values introduced via $\mathsf{inc}_{\mathsf{noop}}$ for *earlier* splits.

Secondly, whenever a constraint is fired in our simulation, it will generate a fresh data value via $\mathsf{inc}_{\mathsf{noop}}$ before some split δ. In an accepting run, the value will be revisited later using $\mathsf{dec}_{\mathsf{push}}$, $\mathsf{dec}_{\mathsf{pop}}$ and $\mathsf{dec}_{\mathsf{noop}}$ (in that order). Referring to the tree underlying a run of $\mathcal{A}_{\mathcal{B}}$, let n_1, n_2 be the nodes corresponding to $\mathsf{dec}_{\mathsf{push}}$ and $\mathsf{dec}_{\mathsf{noop}}$ respectively (n_1 will also correspond to $\mathsf{dec}_{\mathsf{pop}}$). Because of the stack discipline, the least common ancestor of n_1 and n_2 must be some split node δ'.

If δ' is equal to δ then the firing of the constraint is fully compatible with the behaviour of EBVASS in that the data values will be consumed in the left and right branch of the split respectively. However, we could also have $\delta \neq \delta'$ in which case δ will be above δ' in the tree because of scoping. In this case, observe that, by delaying the sequence $\mathsf{dec}_{\mathsf{noop}}/\mathsf{dec}_{\mathsf{push}}, \mathsf{inc}_{\mathsf{noop}}$ (that was originally executed at δ) until just before δ', we will obtain another accepting run. In particular, the use of the least common ancestor guarantees that scoping will be preserved. Consequently, all points at which fresh data values are introduced can be moved into positions where they faithfully represent EBVASS constraints.

□

9 UNDECIDABILITY FOR $\beta \to \beta \to (\beta \to \beta) \to \beta$

Here we show, by reduction from reachability in reset VASS [Araki and Kasami 1976], that observational equivalence in finitary RML is undecidable for closed terms of type $\beta \to \beta \to (\beta \to \beta) \to \beta$. We work with a small-step variant of the machines, given next.

Definition 9.1. A reset VASS is a tuple $(Q, k, q_0, \Delta_i, \Delta_d, \Delta_0)$, where Q is a finite set of states, k is the number of counters, $q_0 \in Q$ is the initial state and $\Delta_i, \Delta_d, \Delta_0 \subseteq_{\mathrm{fin}} Q \times \{1, \cdots, k\} \times Q$ represent the available increment, decrement and reset transitions respectively.

A configuration is a pair $(q, \bar{v})$ where $q \in Q$ is the current state and $\bar{v} \in \mathbb{N}^k$ is the vector of counter values. We define the reset vector operations $\rho_i : \mathbb{N}^k \longrightarrow \mathbb{N}^k$, where $\rho_i(\bar{v})$ agrees with $\bar{v}$ on every position except possibly position i, which it sends to 0. A reset VASS can transition from $(q, \bar{v})$ to $(q', \bar{v}')$ if

- $(q, i, q') \in \Delta_i$ and $\bar{v}' = \bar{v} + \bar{e}_i$, or
- $(q, i, q') \in \Delta_d$ and $\bar{v}' = \bar{v} - \bar{e}_i$, or
- $(q, i, q') \in \Delta_0$ and $\bar{v}' = \rho_i(\bar{v})$.

A run of the VASS is then a sequence of configurations $c_0, c_1, \ldots, c_n$ such that the VASS can transition from c_{i-1} to c_i as above. The *reachability problem* for reset VASS asks, given a reset VASS and target state q_f, whether there is a run from $(q_0, \bar{0})$ to $(q_f, \bar{0})$. The problem is known to be undecidable [Araki and Kasami 1976].

Observe that arenas used for modelling closed terms of type $\beta \to \beta \to (\beta \to \beta) \to \beta$ feature 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 $\begin{tikzcd} q_0 & a_0 \arrow[l, bend right] & q_1 \arrow[l, bend right] & a_1 \arrow[l, bend right] & q_2 \arrow[l, bend right] & a_2 \arrow[l, bend right] & q_3 \arrow[l, bend right] & q_4 \arrow[l, bend right] \end{tikzcd}$. 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 $\begin{tikzcd} q_1 & a_1 \arrow[l, bend right] \end{tikzcd}$, where k is the number of counters. Each segment $\begin{tikzcd} q_1 & a_1 \arrow[l, bend right] \end{tikzcd}$ 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 $\overset{\curvearrowleft}{q_2\ a_2}\overset{\curvearrowleft}{\ q_3}\overset{\curvearrowleft}{\ q_4}$, 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 $\overset{\curvearrowleft}{q_3\ a_3}$, 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 $\overset{\curvearrowleft}{q_2\ a_2}$, 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 $\overset{\curvearrowleft}{q_1\ a_1}$ 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 the following theorem.

Theorem 9.2. *Given a reset VASS $\mathcal{A} = (Q, k, q_0, \Delta_i, \Delta_d, \Delta_0)$ and target state $q_f \in Q$, there are* RML*-terms* $\vdash M, N : \mathsf{unit} \to \mathsf{int} \to (\mathsf{unit} \to \mathsf{int}) \to \mathsf{unit}$ *such that $M \cong N$ iff there is a run of $\mathcal{A}$ reaching configuration $(q_f, \bar{0})$.*

Proof. The RML term that can simulate a given reset VASS according to our representation scheme is given in Fig. 11. It will correspond to M in our argument. As before, the only difference between M and N will be the place where final-state detection takes place, i.e. to obtain N we replace **assert** (!State = $[\![q_f]\!]$) in line 41 with **assert**(False). The code refers to transitions from Δ_i, Δ_d and Δ_0 as $(q, \bar{e}_i, q')$, $(q, -\bar{e}_i, q')$ and $(q, \bar{0}_i, q')$ respectively. We also assume injective functions $[\![-]\!] : Q \to \mathbb{N}$ and $[\![-]\!] : \Delta_i \uplus \Delta_d \uplus \Delta_0 \to \mathbb{N}$. Next we explain how the various variables and assertions inside the term support our simulation.

The simulation is guided by several “global” variables defined at the top level. State corresponds to the state of the rVASS, whereas Run_Stage indicates stages of the simulation. Its initial value is 0 and values 1, 2 are used to induce the initialising segment $q_0a_0q_1a_1q_2a_2q_3q_4$ (lines 11, 25, 40), after which Run_Stage reaches value 3.

Then a segment $\overset{\curvearrowleft}{q_1\ a_1}$ is created for every counter (lines 13-15): Counter_Num is used to keep track of the counter numbers and Active represent the status of the counter, with active meaning “not reset”. After the last counter is handled, Run_Stage is set to 4, and the simulation of transitions can begin.

The choice of increments or resets to fire is done by checking whether O-moves q_2 contain numbers of such transitions (this is done by inspecting y). Deletions are selected in lines 52-54 by

```
let
  State = ref $[\![q_0]\!]$; New_Threads_Allowed = ref True; Expect_Inc = ref False;
  Run_Stage = ref 0;  Awaiting_New_Counter = ref 1
in
  $\lambda x^{\mathsf{unit}}$.
    let
      Counter_Num = ref 0;  Active = ref True
    in
      assert (!Awaiting_New_Counter $\neq$ 0); assert (!New_Threads_Allowed = True);
      if (!Run_Stage = 0) then
        Active := False; !Run_Stage = 1
      else if (!Run_Stage = 3) then
        Counter_Num := !Awaiting_New_Counter; Awaiting_New_Counter++
        if (!Awaiting_New_Counter = d+1) then
          Awaiting_New_Counter := 0; Run_Stage := 4
        else ()
      else
        assert (!Run_Stage = 4);
        Counter_Num := !Awaiting_New_Counter; Awaiting_New_Counter := 0
      $\lambda y^{\mathsf{int}}$.
        let
          Incremented = ref False;  Decremented = ref False
        in
          if (!Run_Stage = 1) then
            Run_Stage := 2; Incremented := True; Decremented := True
          else
            assert(!Run_Stage=4); assert (!Active=True); assert (!Expect_Inc=False);
            assert(!New_Threads_Allowed=True); assert(!Await_New_Counter=0); assert($y \in [\![\Delta_i]\!] \cup [\![\Delta_0]\!]$);
            $\bigcup_{(q,\bar{e_i},q') \in \Delta_i}$:
              if ($y$ = $[\![(q, \bar{e_i}, q')]\!]$) then
                assert (!State = $[\![q]\!]$); assert (!Counter_Num = i);
                State := $[\![q']\!]$; Expect_Inc := True
              else ()
            $\bigcup_{(q,\bar{0_i},q') \in \Delta_0}$:
              if ($y$ == $[\![(q, \bar{0_i}, q')]\!]$) then
                assert (!State = $[\![q]\!]$); assert (!Counter_Num = $i$);
                State := $[\![q']\!]$; Active := False; Awaiting_New_Counter := $i$
            $\lambda f^{\mathsf{unit} \to \mathsf{int}}$.
              if (!Run_Stage = 2) then
                Run_Stage := 3;  f ();
                Run_Stage := 5;  assert (!State = $[\![q_f]\!]$)
              else
                assert (!New_Threads_Allowed = True); assert (!Run_Stage = 4);
                assert (!Active = True); assert (!Await_New_Counter = 0);
                if (!Expecting_Inc = True) then
                  assert (!Incremented = False); Incremented := True; Expecting_Inc := False;
                  f (); Run_Stage := 5; assert (!Decremented = True OR !Active = False);
                else
                  let
                    z = ref 0
                  in
                    New_Threads_Allowed := False;
                    z := f ();
                    New_Threads_Allowed := True;
                    assert ($z \in [\![\Delta_d]\!]$); assert (!Decremented = False); assert (!Active = True);
                    $\bigcup_{(q,-\bar{e_i},q') \in \Delta_d}$:
                      if $z$ == $[\![(q, -\bar{e_i}, q')]\!]$ then
                        assert (State = $[\![q]\!]$); assert (!Counter_Num = $i$);
                        State := $[\![q']\!]$; Decremented := True
                      else ()
```

Fig. 11. The term encoding rVASS reachability

making a special call to f (and halting the whole simulation for the time being with a lock variable New_Threads_Allowed). Because the O-moves are used to let O choose numbers, the term has type unit → int → (unit → int) → unit.

- Increments are handled in lines 27-33. The assertions (lines 27-28) check, among others, whether q_2 is justified by a $\overset{\curvearrowleft}{q_1 \;\; a_1}$ corresponding to an active counter of the right number. State is updated in line 31 and line 32 sets a flag Expect_Inc, which will force $q_3 q_4$ as the next moves. Note the locally declared variables Incremented and Decremented (initially set to False) that will keep track of whether the value has been used up by a decrement. Line 43-47 correspond to $q_3 q_4$, where q_3 must be justified by the preceding a_2 (this is checked by verifying that Incremented is False; in all other segments it will be equal to True). Subsequently, Expect_Inc is reset and Incremented is set to True, and f() (line 47) corresponds to playing q_4.
- Decrements are handled in lines 55-60. First we need to check whether q_3 is really justified by a $\overset{\curvearrowleft}{q_2 \;\; a_2}$ segment corresponding to an active and undecremented counter (line 55). Then line 58 checks whether the right counter is being decremented and whether the transition is consistent with the current state. As a result, State is updated and Decremented is set to True.
- The treatment of resets corresponds to lines 34-37. We check whether the right counter is being reset in line 36 and Active is set to False. In addition, Awaiting_New_Counter is set to the number of the counter in order to signal that a new $\overset{\curvearrowleft}{q_1 \;\; a_1}$ should be created to support future operations on the counter, which is handled by lines 18-19.
- a_4 moves will trigger zero-testing (line 47) and Run_Stage will be changed to 5. During this phase only a_4 moves can be responded to by P and the final such move will be handled by code in line 41, which will test whether a final state has been reached.

Altogether, the strategies corresponding to M and N correspond to simulating the given rVASS. If $(q_f, \bar{0})$ is not reachable then the differences between them do not matter because the zero-testing performed in M will never lead to a complete play. In contrast, if $(q_f, \bar{0})$ is reachable then the insertion of **assert**(False) in line 41 will prevent the strategy for N from generating a complete play that the strategy for M will be able to generate. The result then follows from Theorem 3.5. □

The following is an immediate consequently of the previous result.

Corollary 9.3. *The observational equivalence problem for* RML *terms of type* $\vdash \beta \to \beta \to (\beta \to \beta) \to \beta$ *is undecidable.*

The choices of transitions to fire in the argument above could have been implemented otherwise. Instead of letting O choose numbers explicitly, one can introduce T special segments $\overset{\curvearrowleft}{q_1 \;\; a_1}$, where T is the number of transitions, and O can then select a transition by pointing at one of them with $\overset{\curvearrowleft}{q_2 \;\; a_2}$. This scheme will not require the use of int in the signature of the term and the term will have type unit → unit → (unit → unit) → unit.

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 the fragment $\mathrm{RML}_{\mathrm{VPC}}$, 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.

We anticipate a close connection between VPCMA and DTA# of Jacquemard et al. [Jacquemard et al. 2016]. However, for our purposes (compositional construction of automata from terms) it is much more convenient to rely on VPCMA than DTA#. Arguably, the correspondence between VPCMA and EBVASS is also more direct and intuitive, given that #-stuttering needs to be used for DTA#.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for numerous constructive suggestions and to Ranko Lazić and Sylvain Schmitz for discussions on VASS. Conrad Cotton-Barratt's work was supported by an EPSRC Doctoral Training Grant. Andrzej Murawski's work was partially supported by a Royal Society Leverhulme Trust Senior Research Fellowship (LT170023). Luke Ong's work was partially supported by EPSRC grant EP/M023974/1 and Merton College Research Fund.

REFERENCES

Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. 2000. Full Abstraction for PCF. *Inf. Comput.* 163, 2 (2000), 409–470.

Samson Abramsky and Guy McCusker. 1996. Linearity, Sharing and State: a fully abstract game semantics for Idealized Algol with active expressions. *Electr. Notes Theor. Comput. Sci.* 3 (1996), 2–14.

Samson Abramsky and Guy McCusker. 1997. Call-by-Value Games. In *CSL (Lecture Notes in Computer Science)*, Vol. 1414. Springer, 1–17.

Rajeev Alur and P. Madhusudan. 2004. Visibly pushdown languages. In *STOC*. ACM, 202–211.

Toshiro Araki and Tadao Kasami. 1976. Some Decision Problems Related to the Reachability Problem for Petri Nets. *Theor. Comput. Sci.* 3, 1 (1976), 85–104.

Henrik Björklund and Thomas Schwentick. 2007. On Notions of Regularity for Data Languages. In *FCT (Lecture Notes in Computer Science)*, Vol. 4639. Springer, 88–99.

Henrik Björklund and Thomas Schwentick. 2010. On notions of regularity for data languages. *Theor. Comput. Sci.* 411, 4-5 (2010), 702–715.

Conrad Cotton-Barratt. 2017. *Using Class Memory Automata in Algorithmic Game Semantics*. Ph.D. Dissertation. University of Oxford.

Conrad Cotton-Barratt, David Hopkins, Andrzej S. Murawski, and C.-H. Luke Ong. 2015a. Fragments of ML Decidable by Nested Data Class Memory Automata. In *FoSSaCS (Lecture Notes in Computer Science)*, Vol. 9034. Springer, 249–263.

Conrad Cotton-Barratt, Andrzej S. Murawski, and C.-H. Luke Ong. 2015b. Weak and Nested Class Memory Automata. In *LATA (Lecture Notes in Computer Science)*, Vol. 8977. Springer, 188–199.

Conrad Cotton-Barratt, Andrzej S. Murawski, and C.-H. Luke Ong. 2017. ML and Extended Branching VASS. In *ESOP (Lecture Notes in Computer Science)*, Vol. 8977. Springer, 188–199.

Philippe de Groote, Bruno Guillaume, and Sylvain Salvati. 2004. Vector Addition Tree Automata. In *LICS*. IEEE Computer Society, 64–73.

Dan R. Ghica and Guy McCusker. 2000. Reasoning about Idealized ALGOL Using Regular Languages. In *ICALP (Lecture Notes in Computer Science)*, Vol. 1853. Springer, 103–115.

Benny Godlin and Ofer Strichman. 2009. Regression verification. In *DAC*. ACM, 466–471.

Kohei Honda and Nobuko Yoshida. 1999. Game-Theoretic Analysis of Call-by-Value Computation. *Theor. Comput. Sci.* 221, 1-2 (1999), 393–456.

David Hopkins. 2012. *Game Semantics Based Equivalence Checking of Higher-Order Programs*. Ph.D. Dissertation. Department of Computer Science, University of Oxford.

David Hopkins, Andrzej S. Murawski, and C.-H. Luke Ong. 2011. A Fragment of ML Decidable by Visibly Pushdown Automata. In *ICALP (2) (Lecture Notes in Computer Science)*, Vol. 6756. Springer, 149–161.

J. Martin E. Hyland and C.-H. Luke Ong. 2000. On Full Abstraction for PCF: I, II, and III. *Inf. Comput.* 163, 2 (2000), 285–408.

Florent Jacquemard, Luc Segoufin, and Jerémie Dimino. 2016. FO2(<, +1, ~) on data trees, data tree automata and branching vector addition systems. *Logical Methods in Computer Science* 12, 2 (2016).

Ranko Lazic and Sylvain Schmitz. 2015. Nonelementary Complexities for Branching VASS, MELL, and Extensions. *ACM Trans. Comput. Log.* 16, 3 (2015), 20:1–20:30.

Andrzej S. Murawski. 2005. Functions with local state: regularity and undecidability. *Theoretical Computer Science* 338(1/3) (2005), 315–349.

Andrzej S. Murawski, C.-H. Luke Ong, and Igor Walukiewicz. 2005. Idealized Algol with Ground Recursion, and DPDA Equivalence. In *ICALP (Lecture Notes in Computer Science)*, Vol. 3580. Springer, 917–929.

Andrzej S. Murawski, Steven J. Ramsay, and Nikos Tzevelekos. 2015. Game Semantic Analysis of Equivalence in IMJ. In *Proceedings of ATVA (LNCS)*, Vol. 9364. Springer, 411–428.

Andrzej S. Murawski and Nikos Tzevelekos. 2009. Full Abstraction for Reduced ML. In *FOSSACS (Lecture Notes in Computer Science)*, Vol. 5504. Springer, 32–47.

Andrzej S. Murawski and Nikos Tzevelekos. 2011. Algorithmic Nominal Game Semantics. In *ESOP (Lecture Notes in Computer Science)*, Vol. 6602. Springer, 419–438.

Andrzej S. Murawski and Nikos Tzevelekos. 2012. Algorithmic Games for Full Ground References. In *ICALP (2) (Lecture Notes in Computer Science)*, Vol. 7392. Springer, 312–324.

C.-H. Luke Ong. 2002. Observational equivalence of 3rd-order Idealized Algol is decidable.. In *Proceedings of IEEE Symposium on Logic in Computer Science*. Computer Society Press, 245–256.

Andrew M. Pitts and Ian D. B. Stark. 1998. Operational reasoning for functions with local state. In *Higher order operational techniques in semantics*. Cambridge University Press, 227–273.

John C. Reynolds. 1981. The essence of ALGOL. In *Algorithmic Languages*. North Holland, 345–372.

Received October 2018