Using Class Memory Automata in Algorithmic Game Semantics

Conrad Cotton-Barratt

Balliol College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2016

Abstract

Automata over infinite alphabets are a powerful extension of traditional automata theory, which have begun to be used in solving problems in verification and databases. The infinite nature of the alphabet allows the automata to model potentially unbounded processes or structures at a higher level of precision than finite alphabet automata. Game semantics provides a powerful framework for ascribing fully abstract semantics to higher-order programs, by modelling programs as strategies for interaction between the program and the environment. Questions about properties of programs can then be reduced to checking properties of these strategies.

This thesis aims to continue the classification of the fragments of the higher-order call-by-value language RML for which observational equivalence is decidable. To do so, we will develop new variants of class memory automata that allow us to express and compare languages representing game semantic strategies.

We begin our contributions by looking at class memory automata, a natural form of automata over infinite alphabets. We identify a natural restriction – which we call *weakness* – that leads to better algorithmic and complexity properties. We then admit a tree-structure to the infinite alphabets being used, and present a decidable form of class memory automata over these structured inputs. We also introduce and study a form of automaton combining the class memory operation over infinite alphabets with a visibly pushdown stack.

Next, we use these form of class memory automata in deciding observational equivalence of RML programs. RML can be viewed as the restricton of Standard ML to ground-type references, with a "bad variable" construct. Using the game semantics of RML, we reduce observational equivalence to equivalence of these variants of class memory automata. These methods provide new results on which fragments of RML have decidable observational equivalence. We also present several new undecidability and results, showing fragments of RML to have an undecidable observational equivalence problem.

Acknowledgements

First and foremost I would like to thank my supervisors, Luke Ong and Andrzej Murawski, for their guidance over the course of this degree. They have been generous with their advice and expertise, which has been invaluable.

Without funding from EPSRC this work would never have happened, and I am grateful to them for making this possible.

I'd like to thank my transfer and confirmation examiners, James Worrell and Samson Abramsky, for their comments and suggestions.

I am fortunate and privileged to have many wonderful friends in Oxford and beyond, who have made these four years immensely enjoyable: my thanks for providing a welcome break from work.

Finally, I am grateful and forever indebted to my mother, father, brother, and sister-in-law, for their support, encouragement, and advice that helped me through this degree.

Contents

1 Introduction **11**

1.1 Data languages . . . 11

1.2 Algorithmic Game Semantics . . . 12

1.3 Thesis Overview . . . 13

I Automata Theory **17**

2 Preliminaries **19**

2.1 Vector addition systems with state . . . 19

2.1.1 Reset VASS . . . 20

2.2 Visibly pushdown automata . . . 21

2.3 Data words and automata over data words . . . 22

2.3.1 Class Memory Automata & Data Automata . . . 23

3 Weak Class Memory Automata **29**

3.1 Definition and Basic Properties . . . 29

3.2 Link with Other Automata . . . 32

3.2.1 Weak Data Automata . . . 35

3.3 Link with Logic . . . 36

3.4 Summary . . . 40

4 Class Memory Automata over Nested Data **43**

4.1 Nested Data Values & Class Memory Automata . . . 44

4.2 Nested Data CMA Basic Properties . . . 48

4.3 Link with Nested Data Automata . . . 53

4.4 Summary . . . 55

5 Visibly Pushdown Class Memory Automata **57**
5.1 Definition and Basic Properties 58
5.1.1 Weak VPCMA 60
5.1.2 Closure Properties 61
5.2 Reduction to EBVASS 62
5.2.1 Extended Branching VASS 62
5.2.2 VPCMA - EBVASS reduction 64
5.3 Scoping VPCMA 70
5.3.1 Basic Properties 70
5.3.2 Equivalence with VPCMA 72
5.4 Reduction from EBVASS to SVPCMA 72
5.4.1 Reduction from BVASS 74
5.4.2 Reduction from EBVASS 75
5.5 Summary 77

II Observational Equivalence of RML **79**

6 Preliminaries **81**
6.1 RML 81
6.2 Game Semantics 83
6.2.1 Games, plays, and strategies 85
6.2.2 Game semantics for RML 88
6.3 Observational equivalence 93
6.3.1 Deciding observational equivalence of RML 93

7 Decidable fragments of $\mathrm{RML}_{2\vdash 1}$ **101**
7.1 $\mathrm{RML}_{2\vdash 1}$ 101
7.2 $\mathrm{RML}_{2\vdash 1}^{\mathrm{P-str}}$ 104
7.2.1 Fragment Definition 107
7.2.2 Deciding Observational Equivalence of $\mathrm{RML}_{2\vdash 1}^{\text{P-Str}}$ 107
7.3 $\mathrm{RML}_{2\vdash 1}^{\mathrm{LHS-O-str}}$ 121
7.3.1 Fragment definition 122
7.3.2 Deciding Observational Equivalence 123

7.4 Summary 130

8 Hardness and Undecidability Arguments **133**

8.1 VASS-based reductions 134

8.1.1 VASS Coverability to $\vdash \beta \to \beta \to \beta$ 134

8.1.2 RVASS coverability to $\vdash \beta \to \beta \to \beta \to \beta$ 138

8.1.3 VASS reachability to $\vdash \beta \to (\beta \to \beta) \to \beta$ 141

8.1.4 RVASS reachability to $\vdash \beta \to \beta \to (\beta \to \beta) \to \beta$ 146

8.2 Undecidability at left-hand side types 146

8.2.1 Queue Machines 148

8.2.2 Undecidability at $(\beta \to \beta \to \beta \to \beta) \to \beta \to \beta \to \beta \vdash \beta$ 149

8.3 Summary 158

9 $\mathrm{RML}_{\mathrm{VPCMA}}$ **159**

9.1 $\mathrm{RML}_{\mathrm{VPCMA}}$ fragment definition 160

9.2 Reduction to SVPCMA 162

9.2.1 Encoding pointers 162

9.2.2 Constructions 163

9.3 Reduction from SVPCMA 165

9.4 Summary 170

10 Conclusion **171**

10.1 Class memory automata 171

10.2 RML 172

10.2.1 Open cases 174

Bibliography **177**

Index **184**

Chapter 1

Introduction

1.1 Data languages

A data word is a word over a finite alphabet in which every letter of the word also has an associated value from an infinite domain, which we call the data value. Similarly, a data tree is a tree in which every node also has an associated data value. Data languages, over words and trees, have proved useful in modelling situations arising in verification and database theory. Data words can be used to model unbounded numbers of concurrent communicating processes [14], properties of which can then be verified by verifying properties of the data word. Data trees have proven apt in modelling XML documents, using data values to model attribute values. This approach has yielded positive results in validating XML schema design [12, 44], and in XML query optimization and analysis [74, 85].

Verifying properties of concurrent systems, and properties of XML schemas and queries, thus becomes much easier in the presence of expressive, decidable forms of logic and automata over data words and trees. Finding such automata and logics has therefore become a significant research area in computer science.

Restricting our attention to data words, we find decidable fragments of classical logic have been studied [16, 20], as well as modal logics, particularly a form of LTL with a "freeze" operator [25, 24, 23].

In terms of automata over data words, register automata [47, 75] are natural choice, extending a finite automaton with registers, each of which can store a single data value at a time. Transitions of the register automaton can then depend on the current state and whether the read data value matches the data value stored in any

of the registers.

Several variants extending register automata have also been proposed [89, 90, 9], including adding pushdown behaviour [17, 66]. Many other forms of automata over infinite alphabets have been proposed, including pebble automata [75], variable finite automata [37], data automata [16], and class memory automata [14].

The last of these, class memory automata, operate by remembering a finite piece of information for each data value they read in the run. The transitions can then depend upon, and update, the remembered information for the data value currently being read. These have proved a relatively robust class of automata over data words, with a decidable emptiness problem and closure under union, intersection, and concatenation. In Part I of this thesis we search for variants of class memory automata that increase the expressive power whilst retaining decidability.

1.2 Algorithmic Game Semantics

A denotational semantics aims to assign to program terms a "meaning" that faithfully represents the effect of the term. When the representation of a term – its denotation – is suitably faithful, properties of a program can be verified by verifying properties of its denotation. Of particular interest are fully abstract semantics, which are those where the denotations of two terms are equivalent if and only if those two terms can be substituted for one another in any context without changing any observable behaviour.

Game semantics is a form of denotational semantics that models computation as a dialogue between the program and its environment, and gives the denotation of a term as a strategy of how that program should respond to the environment in these dialogues. Since being first used in modelling programming languages in [42, 3, 76] game semantics has proven to be a powerful paradigm for providing denotational semantics, having been used to provide the first fully abstract semantics for PCF (an implementation of the simply typed lambda calculus with recursion) [43, 2], Idealized Algol (a language similar to PCF with the addition of state and a bad-variable constructor [82]) [5], and various related extensions [4, 5, 1, 7, 52, 58, 64]. It has also been used to provide fully abstract semantics for call-by-value languages with state including RML (the call-by-value equivalent of Idealized Algol) [6, 38], Reduced ML

(like RML but without the bad variable constructor [88]) [67], and several related languages [69, 70, 72]. More recently, it has begun to be used to provide some of the first denotational semantics of object-oriented languages [71].

Algorithmic game semantics seeks to use game semantics to automatically decide properties of programs. It has seen particular use in deciding observational equivalence (OE) – whether two terms can be substituted for one another in all contexts without changing the observable outcome.

This algorithmic use of game semantics proved highly effective in proving decidability of fragments of Idealized Algol (IA). In general observational equivalence is undecidable for any type, since the whole of arithmetic is contained in IA, but it is natural to restrict to a finitary fragment, in which there are only a finite number of values for the integer type. However, restricting to finitary IA is not – by itself – sufficient to guarantee decidability of observational equivalence, since higher-order functions can generate infinite behaviour that leads to undecidability [55].

In [32, 61, 77, 63, 73, 65] it was shown that OE of IA is decidable for terms of order up to and including order 3, and undecidable for all higher-orders. The first of these, in [32], uses regular expressions to express the game semantic strategies of terms. Observational equivalence is thus reduced to equivalence of these regular expressions, which is decidable. Later results [77, 63, 73, 65] used a related approach to construct automata from the IA terms recognising a language that uniquely represented the strategy. Observational equivalence was thus reduced to equivalence of these automata. This idea of reduction from terms to automata representing the game semantic strategy has proved highly effective at deciding observational equivalence, and is central to Part II of this thesis.

1.3 Thesis Overview

The motivating goal of this thesis is to classify the fragments of the call-by-value higher-order language RML into those fragments for which observational equivalence is decidable, and those for which it is not. As described above, algorithmic game semantics provides a powerful approach for achieving this, as the full abstraction results for RML [5, 6] give us a relatively concrete problem to then work on. Following

similar lines to [63, 65, 73, 41, 40], we reduce the problem of equivalence of strategies, to equivalence of automata. In attempting to make these reductions, we found that the forms of automata already studied were insufficient to naturally model the behaviour of the RML strategies. We hence developed new models of automata – in particular new variants of class memory automata (CMA) – to allow us to make these reductions.

Part I of this thesis is therefore pure automata theory, in which we present several new forms of class memory automata, study their algorithmic properties, and examine their relation to existing automata. Once this theoretical groundwork is done, we move to Part II, which focuses on decidability of RML by providing reductions from RML-terms to the automata introduced in Part I.

We begin Part I with background technical material in Chapter 2, covering existing models of computation that we will use throughout the thesis. As well as covering the basics of automata over infinite alphabets, including class memory automata, this chapter includes a brief recap of visibly pushdown automata, and of vector addition systems with state.

The original content of this thesis begins in Chapter 3. This chapter focuses on a restriction of class memory automata – which we call weakness – and examines the resulting improved algorithmic properties of weak CMA. In particular, we find that deterministic weak CMA are closed under Boolean operations, and have an ExpSpace-complete emptiness problem. We find that closely related forms of automata have already been studied in the literature, and clarify these connections. We also examine weak CMA's link with logic, identifying a fragment of first-order logic with which they are equiexpressive.

In Chapter 4 we examine a form of class memory automata operating over nested data values, in which additional tree-structure is given to the data set. We find that these nested data CMA have an undecidable emptiness problem in general, but that with the reintroduction of the weakness constraint from Chapter 3, decidability is recovered. We observe these automata have the same closure properties as normal CMA. We also show that nested data CMA are equivalent to a recently proposed form of data automaton over nested data values [22].

The final chapter of Part I, Chapter 5, proposes a form of class memory automata with a kind of visible stack, that we call visibly pushdown CMA. These do not simply extend CMA with a finite stack, but use the stack to store the data value read when a push-move is made, and use this to enforce that the corresponding pop-move has the same data value. Although unable to show whether emptiness of these automata is decidable, we show that the emptiness problem is equivalent to the reachability problem for extended branching VASS, which were introduced as a minor extension of branching VASS in [45].

We begin Part II with another preliminaries chapter. Chapter 6 gives the syntax and semantics of the language RML, as well as the game semantic model for the language. We look at the problem of observational equivalence, and recap how the problem has already been shown decidable or undecidable for some fragments of RML and several related languages.

In Chapter 7 we consider $\mathrm{RML}_{2\vdash 1}$, the fragment of RML consisting of terms of order 1 with free variables of order at most 2. We consider two subfragments of $\mathrm{RML}_{2\vdash 1}$, $\mathrm{RML}_{2\vdash 1}^{\mathtt{P\text{-}Str}}$ which restrict the free variables to those of arity 1, and $\mathrm{RML}_{2\vdash 1}^{\mathtt{LHS-O-str}}$ which restricts the arguments of the free variables to arity 1. By reductions to nested data CMA, we show both of these fragments to be decidable, though the complexity of these algorithms is limited by the fact that emptiness of nested data CMA is non-primitive-recursive.

We begin Chapter 8 showing that this non-primitive-recursive complexity is necessary, by reducing coverability of reset VASS to observational equivalence of first-order terms. Having shown decidability of some parts of $\mathrm{RML}_{2\vdash 1}$, it is natural to ask whether the whole fragment is. We prove that it is not, by a reduction showing that just a second-order free variable in a term of ground type is sufficient to encode queue machines. We also look at other remaining types for which decidability of observational equivalence is unknown, and prove that for a large class of these types it is undecidable.

The preceding chapters, together with prior work, have narrowed the types of RML terms without free variables for which decidability of observational equivalence is unknown to just types of the shape $o \to (o \to \cdots \to o) \to o$ where o is of ground type. Finally, we examine this type in detail. In Chapter 9 we identify a fragment

containing these types and show observational equivalence is equivalent to emptiness of the visibly pushdown CMA introduced in Chapter 5, and hence equivalent to reachability in extended branching VASS.

Finally, in Chapter 10 we take stock of the results obtained in this thesis, and examine some of the open research questions arising from this work.

Part I

Automata Theory

Chapter 2

Preliminaries

In this chapter we provide background theoretical machinery that we will use throughout this thesis. In particular, we recap vector addition systems with state, visibly pushdown automata, and some of the key aspects of data languages and class memory automata.

General notation. Given set X we write $\mathcal{P}(X)$ for the powerset of X and $\mathcal{P}_{fin}(X)$ for the set of finite subsets of X. Similarly, for sets X and Y we write $X \subset_{fin} Y$ if X is a finite subset of Y.

Given a set X, X^* is the Kleene star of X, the set of finite sequences of elements of X. Generally here X is some kind of alphabet, and we think of elements of X^* as words or strings. In this setting we write ϵ for the empty string. Given a word $w \in X^*$ we write $w(i)$ for the ith letter in the string.

2.1 Vector addition systems with state

Vector addition systems with state (VASS) [39], and their equivalent formalism, Petri nets [78], are powerful and versatile tools for modelling concurrent systems and computation. Their algorithmic properties are well-studied, though some open questions remain, and for this reason they provide a useful formalism for proving decidability and hardness of other systems in the rest of this thesis.

Informally, a VASS consists of a set of counters and a set of states. Transitions of the system may change the current state and/or increment or decrement the various counters. Which transitions are available at any one point depends on the current state, together with the condition that counters may not be decremented below 0.

Definition 2.1.1 ([39])**.** A VASS is a tuple $\langle Q, k, \Delta \rangle$ where Q is a finite set of states, k is the number of counters, or *dimension* of the system, and $\Delta \subset_{fin} Q \times \mathbb{Z}^k \times Q$ is the set of transitions.

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 current vector, or counter values. The VASS can transition from $(q, \bar{v})$ to $(q', \bar{v}')$ just if $(q, \bar{v}' - \bar{v}, q') \in \Delta$. 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.

We will often want a slightly simpler form of VASS, in which each transition can only increment or decrement a single counter by 1. In this case we may separate the set of transitions into increment and decrement transitions, and write it as $\Delta_i \cup \Delta_d$. It is straightforward to see that a full VASS may be simulated by one of these single-increment/decrement VASS. We use the notation $\bar{e_i}$ for the unit vector which is an increment of the ith counter, leaving all other counters unchanged. $\bar{0}$ denotes the zero vector.

Reachability and Coverability. We will primarily be interested in two decision problems of VASS, reachability and coverability, that we describe here.

The *reachability problem* for VASS asks, given a VASS, an initial state q_0, and target state q_t, whether there is a run of the VASS from initial configuration $(q_0, \bar{0})$ to configuration $(q_t, \bar{0})$. This problem was shown to be ExpSpace-hard in [54], and decidable in [49, 57], though in non-primitive recursive space. The exact complexity of the problem remains an open question, though the ExpSpace lower bound remains the best proven, and all known algorithms require non-PR time [28, 83].

The *coverability problem* for VASS asks, given a VASS, initial state q_0, and target state q_t as above, whether there is a run of the from the initial configuration $(q_0, \bar{0})$ to any configuration $(q_t, \bar{v})$ where $\bar{v} \in \mathbb{N}^k$. Seemingly easier than the reachability problem, this was shown to be ExpSpace-complete in [81].

2.1.1 Reset VASS

Reset VASS (rVASS) are an extension of VASS to allow transitions that reset particular counters to 0. We provide a simplified definition of rVASS that permits only transitions that increment one counter, decrement one counter, or reset one counter.

Definition 2.1.2. For fixed $k \in \mathbb{N}$ we define the symbols 0_i for each $i \leq k$ and write R_k for the set of these symbols. We define the reset vector operations $\rho_i : \mathbb{N}^k \to \mathbb{N}^k$, where $\rho_i(\bar{v})$ agrees with $\bar{v}$ on every position except possibly position i, which it sends to 0.

An rVASS is a triple $(Q, k, \Delta_i \cup \Delta_d \cup \Delta_0)$ where $(Q, k, \Delta_i \cup \Delta_d)$ is a VASS, and $\Delta_0 \subseteq Q \times R_k \times Q$.

A configuration of an rVASS is a pair $(q, \bar{v})$ just as for a VASS. The transitions for an rVASS are the same as in the VASS case except the rVASS in configuration $(q, \bar{v})$ can also follow a transition $(q, 0_i, q')$ to configuration $(q', \rho_i(\bar{v}))$.

Reachability and Coverability. The reachability and coverability problems ask the same questions as in the VASS case. However, for rVASS the complexity is increased: reachability becomes undecidable [11, 84]; coverability remains decidable but its complexity becomes non-primitive recursive [27].

2.2 Visibly pushdown automata

Visibly pushdown automata, introduced in [10], are a restriction of pushdown automata to the case in which the stack operation is determined by the input letter. These define a class of languages that extend regular languages, preserve good closure properties - such as intersection, union, complementation - and maintain a decidable equivalence problem. The cost of this extension is that the complexity of equivalence checking rises to ExpTime [10].

We present the definition of these automata without further remark.

Definition 2.2.1. A *pushdown alphabet* is a triple $\Sigma = (\Sigma_{\texttt{push}}, \Sigma_{\texttt{pop}}, \Sigma_{\texttt{noop}})$, where $\Sigma_{\texttt{push}}$, $\Sigma_{\texttt{pop}}$, and $\Sigma_{\texttt{noop}}$ are disjoint finite sets of symbols. By abuse of notation we may write Σ to mean $\Sigma_{\texttt{push}} \cup \Sigma_{\texttt{pop}} \cup \Sigma_{\texttt{noop}}$, and we may write $a_{\texttt{push}}$, $a_{\texttt{pop}}$, $a_{\texttt{noop}}$ for elements of $\Sigma_{\texttt{push}}$, $\Sigma_{\texttt{pop}}$, $\Sigma_{\texttt{noop}}$ respectively.

A *visibly pushdown automaton* (VPA) is a tuple $\langle Q, \Sigma, \Gamma, \Delta, q_0, F \rangle$ where Q is a finite set of states, Σ is a pushdown alphabet, Γ is a finite stack alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and Δ is the transition relation where:

$$\Delta \subseteq (Q \times (\Sigma_{\texttt{push}} \cup \Sigma_{\texttt{pop}}) \times \Gamma \times Q) \cup (Q \times \Sigma_{\texttt{noop}} \times Q)$$

A configuration is a pair (q, S) where $q \in Q$ is the current state and $S \in \Gamma^*$ is the stack. The initial configuration is (q_0, ϵ). A run of the automaton on $w \in \Sigma^n$ is a sequence of configurations $(q_0, \epsilon), (q_1, S_1), \ldots, (q_n, S_n)$ where for each $1 \leq i \leq n$:

- if $w(i) \in \Sigma_{\texttt{noop}}$ then $(q_{i-1}, w(i), q_i) \in \Delta$ and $S_i = S_{i-1}$; else
- if $w(i) \in \Sigma_{\texttt{push}}$ then there is a $\gamma \in \Gamma$ such that $(q_{i-1}, w(i), \gamma, q_i) \in \Delta$ and $S_i = \gamma \cdot S_{i-1}$; else
- if $w(i) \in \Sigma_{\texttt{pop}}$ then $S_{i-1} = \gamma \cdot S_i$ and $(q_{i-1}, w(i), \gamma, q_i) \in \Delta$.

The run is accepting just if ends in a configuration of the form (q_f, ϵ) where $q_f \in F$.

2.3 Data words and automata over data words

Languages over infinite alphabets have been a relatively recent branch of study in computer science, as it has been realised that they have applications to problems in verification and database theory. For example, an infinite alphabet permits fine-grained modelling of a system of an unbounded number of concurrent processes: each process having its own name as part of the alphabet. Having expressive whilst decidable models of logic and automata over these alphabets allows properties of the modelled system to be verified.

Definition 2.3.1. Formally, we fix a finite alphabet Σ and an infinite set of *data values*, $\mathcal{D}$. We do not assume any additional structure on the data values, so the only test we may perform for $d, d' \in \mathcal{D}$ is whether $d = d'$. We then define a *data alphabet* $\mathbb{D} = \Sigma \times \mathcal{D}$. In the normal fashion *data words* are then elements of $\mathbb{D}^*$, and *data languages* are subsets of $\mathbb{D}^*$.

Notation. We will write (a, d) for elements of $\mathbb{D}$, where $a \in \Sigma$ and $d \in \mathcal{D}$. We will use the standard projection functions $\pi_\Sigma : \mathbb{D} \to \Sigma$ and $\pi_\mathcal{D} : \mathbb{D} \to \mathcal{D}$, and we extend these functions in the natural way to $\mathbb{D}^*$. We may refer to $\pi_\Sigma(w)$ as the *string projection* of w.

The data values of a data word may be thought of as partitioning the positions of the word into data classes. That is, two positions in the word $w(i)$ and $w(j)$ are in the same data class iff $\pi_\mathcal{D}(w(i)) = \pi_\mathcal{D}(w(j))$, and we write this $w(i) \sim w(j)$. It is

obvious that $\sim$ is an equivalence relation, and we call the equivalence classes of the relation *data classes*.

Example 2.3.1. Consider a system consisting of a non-zero and unbounded number of processes, each of which must perform a single action, a, both before and after some global event e. Identifying each process by a data value, the traces of this system will have the form $a^n e a^n$ where each data value appears once on each side of the e event (and the event e gets its own data value). Hence, if we take our dataset to be $\mathbb{N}$ we have words such as:

$$(a,1)(a,2)(e,3)(a,1)(a,2) \quad \text{and} \quad (a,5)(a,2)(a,17)(e,9)(a,2)(a,17)(a,5)$$

We will call the language of all such data words L_e.

2.3.1 Class Memory Automata & Data Automata

This thesis is concerned with variants of *class memory automata* (CMA), and we introduce these here. We also introduce CMA's close relative, *data automata* (DA).

2.3.1.1 Data automata

Data automata were introduced in [16], and are a robust class of automata over data words. However, they are an atypical form of automaton, composed of a transducer that first runs over the input, and then a NFA that runs over substrings of the transducer's output. Informally, the transducer runs on the string projection of the input word w to give output v. Then v is partitioned into subwords corresponding to the data classes of w, and each of those subwords must be accepted by the NFA.

Definition 2.3.2 ([16, 14])**.** A *data automaton* (DA) is a pair $(\mathcal{A}, \mathcal{B})$ where $\mathcal{A}$, which we call the base automaton, is a non-deterministic letter-to-letter string transducer with input alphabet Σ and output alphabet Γ, and $\mathcal{B}$, which we call the class automaton, is a finite automaton with input alphabet Γ.

A word $w \in \mathbb{D}^*$ is accepted by the automaton just if there is a run of $\mathcal{A}$ on $\pi_\Sigma(w)$ with output $v \in \Gamma^*$ such that for each data value d occurring in w the following holds: if $i_1 < \cdots < i_k$ are the positions in w with data value d, then $v(i_1) \ldots v(i_k)$ is accepted by $\mathcal{B}$.

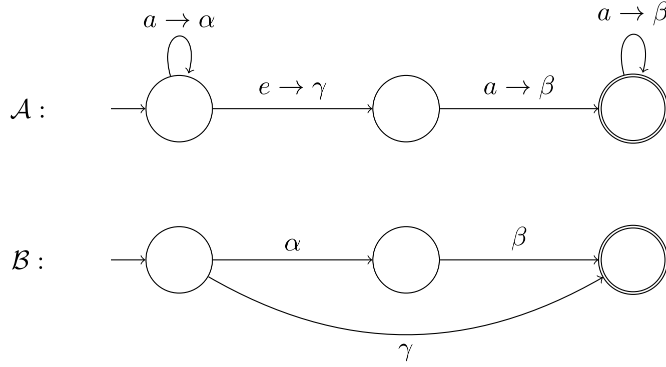


Figure 2.1: A data automaton recognising L_e

Example 2.3.2. We give an example data automaton $(\mathcal{A}, \mathcal{B})$ that recognises the language L_e introduced in Example 2.3.1.

Informally, the class automaton will be able to check that each data value appears twice (or once if the event e), but cannot by itself check that each data value appears once before and once after e. This is where the string transducer comes in: it will give different outputs for the as before and after the event e, and so the class automaton will be able to check each happens once.

Formally, we use transducer output alphabet $\{\alpha, \beta, \gamma\}$, and then have base and class automata as shown in Fig. 2.1.

Emptiness. Emptiness of data automata was shown decidable in [16] by reduction to multicounter automata, whose emptiness problem is known to be equivalent to VASS reachability [33].

2.3.1.2 Class memory automata

Due to their unusual two-stage nature, data automata have no notion of determinism, and are less amenable to algorithmic inductive manipulation. Class memory automata [14] are an expressively equivalent alternative that address these drawbacks.

A class memory automaton operates over data words by remembering each data value it reads, together with the state the automaton was in when it last read that data value. This means the configurations of the automaton include an unbounded memory component. The transitions of the automaton can then depend, as well as on the current state, on the this “remembered state” of the being-read data value. For a run to be accepted the final configuration must meet two criteria: the *global*

acceptance condition requires that a final state has been reached; while the *local acceptance condition* requires that each data value was last seen in the "good" state. Formally:

Definition 2.3.3. We will use a distinguished symbol, $\perp$, to represent fresh data values, and given a set S we write $S_\perp$ for $S \uplus \{\perp\}$.

A *class memory automaton* (CMA) [14] is a tuple $\langle Q, \Sigma, q_0, \delta, F_L, F_G \rangle$ where Q is a finite set of states, Σ is a finite alphabet, $q_0 \in Q$ is the initial state, $F_G \subseteq F_L \subseteq Q$ are sets of globally- and locally-accepting sets (respectively), and δ is the transition map

$$\delta : Q \times \Sigma \times Q_\perp \to \mathcal{P}(Q)$$

The automaton is deterministic if each set in the image of the transition function is a singleton. We will sometimes call the element of the transition domain from $Q_\perp$ the *signature* of the transition. Notationally, if $q' \in \delta(q, a, s)$ we may write this as $q \xrightarrow{a,s} q'$.

A *class memory function* (CMF) is a map $f : \mathcal{D} \to Q_\perp$ such that $f(d) \neq \perp$ for only finitely many $d \in \mathcal{D}$. Given a CMF f, $d \in \mathcal{D}$, and $q \in Q$ we write $f[d \mapsto q]$ for the CMF which agrees with f on all elements of $\mathcal{D}$ except possibly d, which it maps to q.

A configuration of a class memory automaton is a pair (q, f) where $q \in Q$ and f is a class memory function. The initial configuration is (q_0, f_0) where $f_0(d) = \perp$ for every $d \in \mathcal{D}$. A configuration (q, f) is accepting just if it meets the global acceptance condition: $q \in F_G$; and the local acceptance condition: $f(d) \in F_L \cup \{\perp\}$ for every data value d. A run of the automaton on $w \in \mathbb{D}^*$ is a sequence of configurations $(q_0, f_0), (q_1, f_1), \ldots, (q_n, f_n)$ such that for each $1 \leq i \leq n$: if $w(i) = (a_i, d_i)$, then $q_i \in \delta(q_{i-1}, a_i, f_{i-1}(d_i))$ and $f_i = f_{i-1}[d_i \mapsto q_i]$

In the normal way, a data word w is accepted by the automaton just if there is a run of the automaton on w ending in an accepting configuration, and the language recognised by the automaton is the set of words accepted by the automaton. Given an automaton $\mathcal{A}$, we denote this language $\mathcal{L}(\mathcal{A})$.

Note although we have presented CMA as having a transition function δ, it is also perfectly possible to define them in terms of a transition relation $\Delta \subseteq Q \times \Sigma \times$

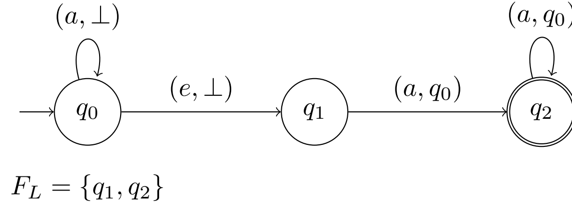


Figure 2.2: A class memory automaton recognising L_e

$Q_\bot \times Q$ instead. The two presentations are obviously equivalent, and we use them interchangeably as convenient.

Example 2.3.3. We give an example CMA recognising the language L_e from Example 2.3.1. The automaton is presented in Fig. 2.2. Note that transitions between states are labelled by pairs of elements: an input letter from Σ and the signature of the transition from $Q_\bot$.

The automaton first reads some number of a-letters with fresh data values whilst staying in state q_0. This means the CMF will hold q_0 as its memory for all these data values. After it has read the global event e, it can then only read a-letters with data values last seen in q_0. This ensures each process is started before the event. The local acceptance condition then enforces the requirement that each such process is read again after the event (and it can only be read once, because once read the CMF will no longer store its state as q_0).

CMA were introduced as a conceptually cleaner alternative to data automata. We sketch the proof, from [14], that CMA and data automata are expressively equivalent, with PTime reductions between them.

Proposition 2.3.1 ([14])**.** *CMA and DA are expressively equivalent, with* PTime *translations between them.*

Proof reproduced from [14]. To simulate a DA $(\mathcal{A}, \mathcal{B})$ with a CMA $\mathcal{C}$, the CMA will simultaneously simulate the run of the transducer with all the runs of the class automaton. This can be done with a state set $Q_\mathcal{C} = Q_\mathcal{A} \times Q_\mathcal{B}$. A configuration of $((q, q'), f)$ then represents that $\mathcal{A}$ is in state q and for each data value d, the run of $\mathcal{B}$ on the class of d is in the state corresponding to the second component of $f(d)$. Formally, if $q \xrightarrow{a} (q', \alpha)$ is in $\mathcal{A}$ and $p \xrightarrow{\alpha} p'$ is in $\mathcal{B}$, we have the transition

$(q,x) \xrightarrow{a,(y,p)} (q',p')$ for every $x \in Q_{\mathcal{A}}$ and $y \in Q_{\mathcal{B}}$. The locally accepting states are those where the second component is accepting in $\mathcal{B}$, and the globally accepting states are those where the first component is also accepting in $\mathcal{A}$.

To simulate a CMA $\mathcal{C}$ with a DA $(\mathcal{A},\mathcal{B})$ the base automaton of the DA will attempt to simulate $\mathcal{C}$ by itself, but guessing the value of $f(d)$ at each point, and saving this guess to the output. The class automaton will then verify these guesses. Formally $Q_{\mathcal{A}} = Q_{\mathcal{C}}$, and for each transition $q \xrightarrow{a,s} q'$ in $\mathcal{C}$ there is a transition $q \xrightarrow{a} (q',(s,q'))$. $\mathcal{B}$ then checks that the first letter in each of the output subwords starts with a letter with first component $\perp$, that the second component of each letter matches the first component of the next letter, and that the second component of the final letter is a locally accepting state.

$\square$

CMA with Labels. We note that the key aspect of the class memory function is that codomain is finite. By means of a product construction it is straightforward to see that the power of CMA is not changed by substituting an arbitrary finite set of labels, L. In this case the transition function becomes of the form:

$$\delta : Q \times \Sigma \times L_\perp \to \mathcal{P}(Q \times L)$$

and the local acceptance condition uses a set $F_L \subseteq L$. In this case when in configuration (q,f) and reading input (a,d), the automaton can transition to $(q', f[d \mapsto l'])$ iff $(q',l') \in \delta(q,a,f(d))$. When using these CMAs with labels we may write $(q',l') \in \delta(q,a,l)$ as $q \xrightarrow{a,l} q' : l'$.

Closure properties. By adapting the standard proofs for DFA it is straightforward to show that CMA are closed under intersection, union, and concatenation. It was shown in [14] that they are not closed under complementation or Kleene star. It was also shown that deterministic CMA, whilst closed under intersection, are not closed under any of union, concatenation, complementation, or Kleene star.

Emptiness. Emptiness of CMA was first shown decidable by the above reduction to data automata ([14]). We here present direct reductions between CMA emptiness and VASS reachability, as the ideas of the proof will be helpful in later chapters.

Proposition 2.3.2. *Emptiness of class memory automata is equivalent to VASS reachability (with both reductions in* PTime*).*

Proof sketch. **Reduction from CMA to VASS.** The key idea of this reduction is using counters to simulate the class memory function. For CMA, in determining whether there is a run from the current configuration to an accepting configuration, the only necessary information is the current state and the number of data values last seen in each state.

Hence, the VASS will have states the same as the CMA's and a counter corresponding to each state, counting the data values "in" that state. For each transition of the class memory automaton $q \xrightarrow{a,s} q'$ the VASS will have a transition from state q to q' decrementing the counter corresponding to s (assuming $s \neq \perp$) and incrementing the counter corresponding to q'. The local acceptance condition is simulated by permitting decrements of the counters corresponding to locally accepting states at any point, and non-emptiness of the CMA is then equivalent to reachability, from $(q_0, \bar{0})$, of a configuration $(q_F, \bar{0})$ where $q_F \in F_G$.

Reduction from VASS to CMA. Given a VASS $\mathcal{V} = (Q, k, \Delta_i \cup \Delta_d)$, initial state $q_0 \in Q$, and target state $q_F \in Q$, we construct a CMA with labels, $\mathcal{A}$, whose language is non-empty iff the the VASS configuration $(q_F, \bar{0})$ is reachable from $(q_0, \bar{0})$.

The key idea here is to store the value of a counter i as the number of data values with label i. The rest of the construction is straightforward, as the state of the $\mathcal{V}$ can just be stored in the state of $\mathcal{A}$. The only subtlety is that an additional label for those increments that have been decremented must be used, and this will be the only locally accepting label. This ensures that all increments have been decremented at the end of the run.

Formally $\mathcal{A}$ will have states Q, initial state q_0, and a set of labels $L = \{1, \ldots, k\} \uplus \{dec\}$. The finite alphabet will just be $\Sigma = \{a\}$. The transition relation will then be as follows: for each increment transition $(q, \bar{e}_i, q')$ we have $q \xrightarrow{a,\perp} q' : i$, and for each decrement transition $(q, -\bar{e}_i, q')$ we have $q \xrightarrow{a,i} q' : dec$. There are no other transitions. The globally accepting states are just $\{q_F\}$ and the locally accepting labels are just $\{dec\}$. □

Chapter 3

Weak Class Memory Automata

We have seen class memory automata (CMA) have both a local and global acceptance condition. The global condition is an "easy" check that the automaton ends in a globally accepting state, while the local condition requires a check for each data value seen: a potentially unbounded number. In this chapter we drop this computationally expensive local acceptance condition, yielding *weak* class memory automata (WCMA). We find that the result is a natural, robust class of automaton, with a lower computational complexity for emptiness checking, and slightly nicer closure properties. In Section 3.2 we show that WCMA are equivalent to several kinds of automata previously independently studied in the literature. Then in Section 3.3 we identify a fragment of existential monadic second-order logic with which WCMA are equivalent.

The results in Sections 3.1 and 3.2, with the exception of Proposition 3.2.5, have been published in [19] and are joint work with Andrzej Murawski and Luke Ong.

3.1 Definition and Basic Properties

Definition 3.1.1. A class memory automata $\langle Q, \Sigma, \delta, q_0, F_L, F_G \rangle$ is *weak* just if $F_L = Q$.

Notationally, when describing a weak class memory automata, only the set of globally accepting states need be provided, so we may write $\langle Q, \Sigma, \delta, q_0, F \rangle$ for $\langle Q, \Sigma, \delta, q_0, Q, F \rangle$.

Example 3.1.1. Without the local acceptance condition, the automaton cannot check that every data class satisfies some good property. Hence, for instance, no

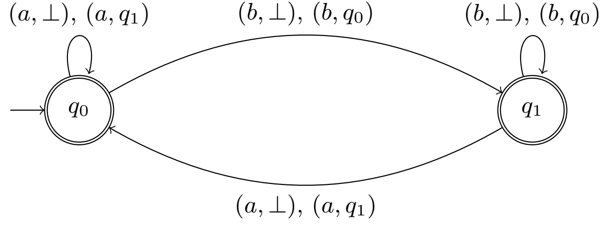


Figure 3.1: A weak class memory automaton recognising L_{ab^*}

WCMA can recognise the same language as the automaton presented in Example 2.3.3, the language L_e.

However, WCMA can still check properties pertaining to each data class so long as that property can be checked as it goes along. To give an example, an automaton recognising the language L_{ab^*} of data words such that each data class alternates between a- and b-positions is presented in Fig. 3.1.

In essence, this automaton remembers for each data value whether it last had an a-position or b-position (remembered by state q_0 or q_1 respectively), and the transitions available enforce that an a-move is only allowed if the read data value was last seen in state q_1 (and hence was a b-move), and vice-versa.

Proposition 3.1.1. *Weak CMA are closed under intersection, union, and concatenation. They are not closed under complementation.*

Proof. Closure under intersection and union can be shown by product constructions. Closure under concatenation can be shown using the same construction as in the NFA case: starting in a set of states as per the first automaton, and non-deterministically guessing when the first word ends (having reached a final state), and switching to the second automaton. The transitions in the second automaton must be amended to allow data values seen during the run of the first automaton to be used as if fresh data values.

To show WCMA are not closed under complementation, we consider the language, $L_{\sim a}$, over $\Sigma = \{a, b\}$ of data words such that for each data value appearing in the word, at least one position with that data value is an a-position. By a pumping argument, we show that the language $L_{\sim a}$ is not recognisable by a WCMA, but that the language $\overline{L_{\sim a}}$, of data words such that at least one data class appearing in the word does not have any a-positions, is recognisable by a WCMA.

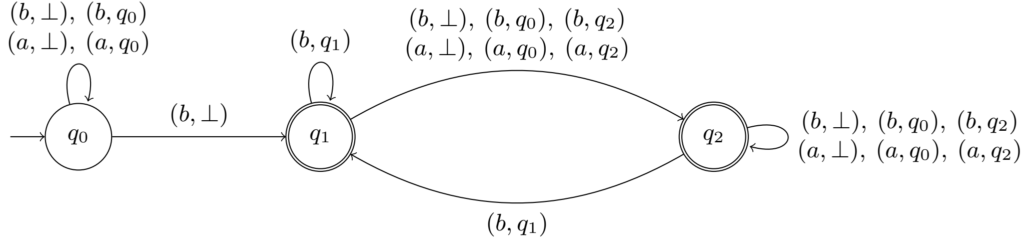


Figure 3.2: A weak class memory automaton recognising $\overline{L_{\sim a}}$

We give a an automaton recognising $\overline{L_{\sim a}}$ in Fig. 3.2. The automaton works by guessing a data value which will not contain any a-positions, and keeping this data value in state q_1.

To show $L_{\sim a}$ is not recognisable by WCMA, suppose there exists a WCMA $\mathcal{A}$ recognising this language, with n states. We note that the word

$$w = (b, d_1)(b, d_2) \dots (b, d_{n+1})(a, d_1)(a, d_2) \dots (a, d_{n+1})$$

(where each d_i is a distinct data value) is in L, and hence has an accepting run on $\mathcal{A}$

$$\rho = (q_0, f_0)(q_1, f_1) \dots (q_{2n+2}, f_{2n+2})$$

By the pigeonhole principle there exists $1 \leq i < j$ such that $q_{n+i} = q_{n+j}$, and it is straightforward to see that

$$\rho' = (q_0, f_0) \dots (q_{n+i}, f_{n+i})(q_{n+j+1}, f'_{n+j+1})(q_{n+j+2}, f'_{n+j+2})(q_{2n+2}, f'_{2n+2})$$

where (for $k > n + j$) $f'_k = f_k[d_x \mapsto f_{n+i}(d_x) : i + 1 \leq x \leq j]$, is is an accepting run of the word

$$w' = (b, d_1) \dots (b, d_{n+1})(a, d_1) \dots (a, d_i)(a, d_{j+1}) \dots (a, d_{n+1})$$

Hence $w' \in \mathcal{L}(\mathcal{A})$, but $w' \notin L_{\sim a}$ ※. □

Proposition 3.1.2. *Deterministic WCMA are closed under all Boolean operations. They are not closed under concatenation.*

Proof. Again, closure under intersection and union can be shown by product constructions. As in the DFA case, the automaton may be complemented by simply complementing the final states.

Non-closure under concatenation follows from the proof that deterministic CMA are not closed under concatenation provided in [14]. The authors of [14] show that if L_1 is the language of all data words ending in an a-position, and L_2 is the language of data words such that no data value appears more than once, then $L = L_1 \cdot L_2$ is not recognisable by a deterministic CMA. It immediately follows that L is not recognisable by a deterministic WCMA, and it is straightforward to see that both L_1 and L_2 are recognised by deterministic WCMA. □

We saw in Section 2.3.1 that emptiness of class memory automata is equivalent to the reachability problem for VASS. By examining the proof of this (first given in [14]) it is apparent that the local acceptance condition corresponds, in the reductions in both directions, to checking that the counters have reached zero. It is therefore a straightforward adaptation of the proof to show that WMCA emptiness is equivalent to VASS state-reachability, or VASS coverability. Hence we get the following result:

Proposition 3.1.3. *The emptiness problem for weak CMA is* ExpSpace*-complete.*

3.2 Link with Other Automata

Class memory automata were introduced (in [14]) as an alternative presentation of data automata. By adapting the proof of equivalence between CMA and DA we find that weakness in CMA corresponds to requiring the class automata of data automata to be prefix-closed ([22]). We also find that WCMA are in fact equivalent to two other existing forms of automata: class counting automata (CCA) and non-reset history register automata (nrHRA). These results are summarised in the main theorem of this section, Theorem 3.2.1.

Theorem 3.2.1. *Weak class memory automata, locally prefix-closed data automata, class counting automata, and non-reset history register automata are all equivalent, with* PTime *translations.*

We prove each of these equivalences in turn, beginning with locally prefix-closed data automata.

Definition 3.2.1 (Decker et al. [22])**.** A data automaton $\mathcal{D} = (\mathcal{A}, \mathcal{B})$ is a locally prefix-closed data automaton (pDA) [22] if all states in $\mathcal{B}$ are final.

If we consider the proof of PTime-equivalence between CMA and DA given in [14], it is straightforward to see that the constructions used transform a WCMA into a pDA and vice-versa. Hence we immediately get the result:

Proposition 3.2.2. *WCMA and pDA are* PTime*-equivalent.*

We now move on to class counting automata.

Definition 3.2.2 (Manuel & Ramanujam [56])**.** A *bag* over $\mathcal{D}$ is a function $h : \mathcal{D} \to \mathbb{N}$ such that $h(d) = 0$ for all but finitely many $d \in \mathcal{D}$. We define $h[d \mapsto n]$ (where $d \in \mathcal{D}$ and $n \in \mathbb{N}$) in the normal way.

Let $C = \{=, \neq, <, >\} \times \mathbb{N}$, which we call the set of constraints. If $c = (\texttt{op}, e) \in C$ and $n \in \mathbb{N}$ we write $n \vDash c$ iff $n \; \texttt{op} \; e$.

A *class counting automaton* (CCA) is a tuple $\langle Q, \Sigma, \Delta, q_0, F \rangle$ where Q is a finite set of states, Σ is a finite alphabet, q_0 is the initial state, $F \subseteq Q$ is the set of accepting states, and Δ, the transition relation, is a finite subset of $Q \times \Sigma \times C \times \{\uparrow^+, \downarrow\} \times \mathbb{N} \times Q$.

A configuration of a CCA, $\mathcal{C} = \langle Q, \Sigma, \Delta, q_0, F \rangle$, is a pair (q, h) where $q \in Q$ and h is a bag. The initial configuration is (q_0, h_0) where h_0 is the zero function. Given a data word $w = (a_1, d_1)(a_2, d_2) \dots (a_n, d_n)$ a run of w on $\mathcal{C}$ is a sequence of configurations $(q_0, h_0)(q_1, h_1) \dots (q_n, h_n)$ such that for all $0 \le i < n$ there is a transition (q, a, c, π, m, q') where $q = q_i$, $q' = q_{i+1}$, $a = a_{i+1}$, $h_i(d_{i+1}) \vDash c$, and

$$h_{i+1} = \begin{cases} h_i[d_{i+1} \mapsto h_i(d_{i+1}) + m] & \text{if } \pi = \uparrow^+ \\ h_{i+1} = h_i[d_{i+1} \mapsto m] & \text{if } \pi = \downarrow \end{cases}$$

The run is accepting if $q_n \in F$.

The *bags* CCA use essentially give a counter for each data value, and the transitions of the CCA can either increment the read data value's counter, or reset it to a fixed number. We can see that WCMA can easily be simulated by CCA by identifying each state with a natural number; then the bag can easily simulate the class memory function by setting the data value's counter to the appropriate number when it is read. To simulate a CCA with a WCMA, we first observe that for any CCA, since counter values can only be incremented or reset, there is a natural number, N, above which different counter values are indistinguishable to the automaton. Thus we need only worry about a finite set of values. This means the value for the counter of

each data value can be stored in the automaton state, and thereby the class memory function. Hence we get:

Proposition 3.2.3. *WCMA and CCA are* PTIME*-equivalent.*

Finally, we consider non-reset history register automata (nrHRA).

Definition 3.2.3 (Tzevelekos & Grigore [90][1])**.** For a positive integer k write $[k]$ for the set $\{1, 2 \ldots, k\}$. Fixing a positive integer m, define the set of labels $\mathsf{Lab} = \mathcal{P}([m])^2$.

A non-reset History Register Automaton (nrHRA) of type m with initially empty assignment is a tuple $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$ where $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and $\delta \subseteq Q \times \Sigma \times \mathsf{Lab} \times Q$.

A configuration of $\mathcal{A}$ is a pair (q, H) where $q \in Q$ and $H : [m] \to \mathcal{P}_{fn}(\mathcal{D})$ where $\mathcal{P}_{fn}(\mathcal{D})$ is the set of finite subsets of $\mathcal{D}$. We call H an *assignment*, and for $d \in \mathcal{D}$ we write $H^{-1}(d)$ for the set $\{i \in [m] : d \in H(i)\}$. The initial configuration is (q_0, H_0), where H_0 assigns every integer in $[m]$ to the empty set.

When the automaton is in configuration (q, H), on reading input (a, d) it can transition to configuration (q', H') providing there exists $X \subseteq [m]$ such that $(q, a, (H^{-1}(d), X), d) \in \delta$ and H' is obtained by removing d from $H(i)$ for each i then adding d to each $H(i)$ such that $i \in X$. A run is defined in the usual way, and a run is accepting if it ends in a configuration (q, H) where $q \in F$.

In [90] the authors already show that nrHRA can be simulated by CMA. Their construction does not make use of the local-acceptance condition, so the fact that nrHRA can be simulated by WCMA is immediate. In order to simulate a given WCMA with state set $[m]$, we take a nrHRA of type m, with place i storing the data values last seen in state i. Thus we get:

Proposition 3.2.4. *WCMA and nrHRA are* PTIME*-equivalent.*

This completes the proof of Theorem 3.2.1. □

[1]We provide a simplified definition to that provided in [90], since we do not need to consider full History Register Automata. In particular, due to Proposition 22 in [90], we need only consider histories, and not registers.

3.2.1 Weak Data Automata

We note that another restriction of data automata, *weak data automata* (WDA), were introduced in [48]. In the rest of this section we briefly show that, despite what the name suggests, these are an entirely different restriction, and are non-comparable with WCMA.

Definition 3.2.4 ([48])**.** WDA are defined using certain kinds of constraints. Given a finite language Γ we consider constraints of three kinds, and in the list below we describe when a data word $w \in \Gamma \times \mathcal{D}$ satisfies a constraint, C (written $w \vDash C$). (Below we assume $\gamma, \gamma' \in \Gamma$ and $R \subseteq \Gamma$.)

- $w \vDash key(\gamma)$ iff every two γ-positions in w have different data values.
- $w \vDash V(\gamma) \subseteq \bigcup_{\gamma' \in R} V(\gamma')$ iff each data class in w with a γ-position also have a γ'-position for some γ' from R.
- $w \vDash V(\gamma) \cap V(\gamma') = \emptyset$ iff no data class in w has both a γ-position and a γ'-position.

A WDA is a pair $(\mathcal{A}, \mathcal{C})$ where $\mathcal{A}$ is a letter-to-letter string transducer with output language Γ, and $\mathcal{C}$ is a set of constraints of the forms above over the language Γ. A data word $w \in (\Sigma \times \mathcal{D})^*$ is accepted by $(\mathcal{A}, \mathcal{C})$ iff there is an accepting run of $\mathcal{A}$ on $\pi_\Sigma(w)$ with output $\hat{w}$ such that if w' is the data word in $(\Gamma \times \mathcal{D})^*$ obtained by attaching w's data values to $\hat{w}$, $w' \vDash C$ for each $C \in \mathcal{C}$.

It was shown in [48] that WDA are a restriction of data automata.

Proposition 3.2.5. *Weak Class Memory Automata and Weak Data Automata are non-comparable.*

Proof. WDA were shown in [48] to be equiexpressive with $\text{EMSO}^2(\sim, +1)$. Consider the language $L_{\sim a}$ introduced in the proof of Proposition 3.1.1. This language is defined by the $\text{FO}^2(\sim, +1)$-formula $\forall x \exists y.(x \sim y \wedge a(y))$, so is recognised by a WDA. However, in proving Proposition 3.1.1 we showed that $L_{\sim a}$ is not recognisable by WCMA. Hence WDA $\nsubseteq$ WCMA.

To show the other non-inclusion, we consider the language L_{a*b}, defined in [48] as the data language over $\Sigma = \{a, b\}$ such that each a-occurrence is followed two places

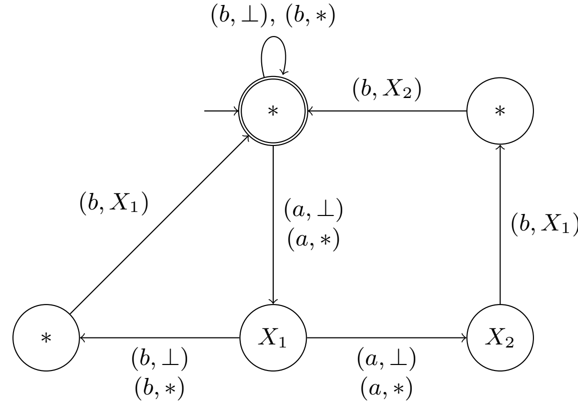


Figure 3.3: A weak class memory automaton recognising L_{a*b}

later by a b-occurrence with the same data value, and the intervening letter is not a b-occurrence sharing the same data value. In the same paper, it was shown that the language L_{a*b} could not be recognised by a WDA. A WCMA recognising this language is presented in Fig. 3.3. (To aid readability of this figure, a single label of $*$ is used for three states, as we do not need the class memory function to distinguish between them.)

□

3.3 Link with Logic

Data automata were first introduced in [16] as a tool to help show the decidability of two-variable logic on data words. It was shown in [16] that data automata (and hence CMA) are equivalent to $\text{EMSO}^2(\sim, <, +1)$: existential monadic second-order logic with two first-order variables. We ask what effect the weak restriction has on the logical expressivity of CMA.

Without the local acceptance condition, the ability of class memory automata to check properties of "later" behaviour in a data word is significantly reduced. In terms of logical expressivity, this corresponds to restrictions on being able to check existence of features "later" in the word than the position currently being considered. To capture this restriction, we introduce the notion of *bounded* quantification. We call (first-order) existential quantification of the simple form $\exists x.\phi(x)$ *unbounded* existential quantification, and we call quantification of the $\exists x \leqslant y.\phi(x, y)$ *bounded* existential quantification, as in this case the variable y is bounding the possible values of x. In

this section we show that WCMA correspond to the fragment of EMSO^2 in which unbounded existential quantification is only allowed at the top level of quantification, and all other existential quantifications must be bounded by the other first-order variable.

Formally, we define this restriction on EMSO as follows:

Definition 3.3.1. Given a finite alphabet Σ and a set of relations X, the *existentially bounded* fragment of existential monadic second-order logic with non-logical symbols $\Sigma \cup X$ (written ∃B-EMSO(X)) consists of formulae ϕ given by the following grammar:

$$\phi ::= \varphi \mid \exists P.\phi \qquad \varphi ::= \psi \mid \exists \nu.\psi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$$

$$\psi ::= \chi \mid \forall \nu.\psi \mid \exists \nu \leqslant \nu'.\psi \mid \psi \wedge \psi \mid \psi \vee \psi$$

$$\chi ::= \nu \in P \mid R(\nu, \ldots, \nu) \mid a(\nu) \mid \neg\chi \mid \chi \wedge \chi \mid \chi \vee \chi$$

Where R is a predicate in X, $a \in \Sigma$, $\nu \neq \nu'$ are first-order variables, and P is from an infinite set of unary predicate variables. The semantics are mostly standard, with first-order variables ranging over positions in the word [16], and $w, s \vDash \exists \nu \leqslant \nu'.\chi$ iff there exists $j \leq s(\nu')$ such that $w, s[\nu \mapsto j] \vDash \chi$.

With this definition, we can prove the equivalence between weak CMA and logic. The only additional predicates we will use are $+1$ and $\oplus 1$. $+1$ is the standard successor relation, where $w, v \vDash x = y + 1$ iff $v(x) = v(y) + 1$. $\oplus 1$ is the class successor relation, where $w, v \vDash x = y \oplus 1$ iff $v(y) < v(x)$, $w(v(x)) \sim w(v(y))$, and if $v(y) < i < v(x)$ then $w(i) \not\sim w(v(x))$.

Theorem 3.3.1. *Weak CMA are equiexpressive with* ∃B-$EMSO^2(+1, \oplus 1)$.

Proof. That Weak CMA can be simulated by ∃B-$\mathrm{EMSO}^2(+1, \oplus 1)$ is a straightforward adaptation of the standard simulation of word automata by EMSO. In particular, for an automaton $\mathcal{A} = \langle \{1, \ldots, n\}, \Sigma, \delta, 1, F \rangle$, the $\mathcal{L}(\mathcal{A})$ is recognised by the formula

$$\exists Y_1 \ldots \exists Y_n.\, (p(Y_1, \ldots, Y_n) \;\wedge\; (\mathit{zero} \in Y_1) \;\wedge\; (\mathit{last} \in F) \;\wedge\; \Phi_\delta)$$

where:

$$
\begin{aligned}
p(Y_1,\dots,Y_n) &\equiv \Big(\forall x. \bigvee_{0<i\leqslant n} x \in Y_i\Big) \wedge \big(\forall x. \bigwedge_{0<i<j\leqslant n} \neg(x \in Y_i \wedge x \in Y_j)\big) \\
zero \in Y_1 &\equiv \exists x.(x \in Y_1 \wedge \forall y. x \neq y+1) \\
last \in F &\equiv \exists x.\Big((\bigvee_{i\in F} x \in Y_i) \wedge \forall y. y \neq x+1\Big) \\
\Phi_\delta &\equiv \forall x. \forall y. y = x+1 \rightarrow \Big(\bigvee_{(i,a,s,K)\in\delta} x \in Y_i \wedge a(x) \wedge (\bigvee_{k\in K} y \in Y_k) \wedge \Phi_{s,x}\Big)
\end{aligned}
$$

where $\Phi_{s,x} \equiv \begin{cases} \exists y \leqslant x.(x = y \oplus 1 \wedge y \in Y_s) & \text{if } s \in \{1,\dots,n\} \\ \forall y. x \neq y \oplus 1 & \text{if } s = \bot \end{cases}$

The proof of the other direction is an adaptation of the proof in [16] for reducing data automata to EMSO^2. First we note that each sentence of $\exists$B-EMSO^2 can be written in an alteration of Scott Normal Form, as a disjunction of clauses of the form:

$$\exists R_1,\dots,R_k(\forall x \forall y.\chi \wedge \bigwedge_{i=1}^{m} \forall x \exists y \leqslant x.\chi_i \wedge \bigwedge_{j=1}^{n} \exists x.\psi_j)$$

where χ, each χ_i and each ψ_j is a quantifier-free formula in at most two variables (by simple adaptation of the proof in [36]). It is then enough to check that each of the clauses of this form can be simulated by a Weak CMA, which can be done with slight alterations to the proof in [16]. We sketch these proofs in Lemmas 3.3.2 to 3.3.4.

As in [16], we will use an automaton, $\mathcal{D}_+$, to recognise the predicate S_{+1}, which identifies whether neighbouring positions have the same data value. The ideas in [15] for showing that unmarked data automata can simulate marked data automata descend directly to locally prefix-closed data automata, and hence we can construct $\mathcal{D}_+$ in this weaker setting. □

Lemma 3.3.2. *For each FO$^2(+1,\oplus 1)$ formula $\phi = \exists x.\chi$ with χ quantifier-free, an equivalent WCMA can be constructed.*

Proof. As χ only has one variable, this becomes a straightforward check for existence of a suitable input letter. □

Lemma 3.3.3. *For each FO$^2(+1,\oplus 1)$ formula $\phi = \forall x \forall y.\chi$ with χ quantifier-free, an equivalent WCMA can be constructed.*

Proof. As in [16], we can rewrite ϕ as a conjunction of formulae of the form:

$$\psi = \forall x \forall y((\alpha(x) \wedge \beta(y) \wedge \delta(x,y)) \rightarrow \gamma(x,y))$$

where α and β denote conjunctions of unary predicates and their negations, $\delta(x,y)$ is wlog either $x = y \oplus 1$ or $x \neq y \oplus 1$, and $\gamma(x,y)$ is a disjunction of atomic formulae or their negations using only the predicate $+1$. In the case that $\delta(x,y) = (x = y \oplus 1)$ it is straightforward, using $\mathcal{D}_+$, to construct a suitable pDA. In the case that $\delta(x,y) = (x \neq y \oplus 1)$ a further analysis on possible $\gamma(x,y)$ yields the options:

$$\textit{false}, \quad x = y+1, \quad y = x+1, \quad x \neq y+1, \quad y \neq x+1, \quad x = y+1 \vee y = x+1$$

- if $\gamma(x,y) \equiv \mathit{false}$, this means that either there are no α-positions, or no β-positions, or precisely one α- and one β-position, such that the α position is the class successor of the β-position. Each of these disjunctions is easily checked by a WCMA.

- if $\gamma(x,y) \equiv x = y+1$, this means either the conditions are never met (and we are as in one of the false settings above), or there is one α-position which is the successor of a β-position and possibly the class successor of another β-position, with no other β-positions, or there is one β-position, and an α-position as its successor, and possibly another α-position as its class successor. Again, this is straightforward to encode in a single-pass WCMA.

- if $\gamma(x,y) \equiv y = x+1$, the cases are very similar to the $x = y+1$ case.

- if $\gamma(x,y) \equiv x \neq y+1$, this means every β-position's successor is either not an α-position or is in the same data class. This is straightforward to check, using $\mathcal{D}_+$.

- if $\gamma(x,y) \equiv y \neq x+1$, this means every α-position's successor is not a β-position. This is straightforward to check, without even looking at data values.

- if $\gamma(x,y) \equiv x = y+1 \vee y = x+1$, the cases are a combination of the $x = y+1$ and $y = x+1$ cases, and easily checked.

$\square$

Lemma 3.3.4. *For each $FO^2(+1, \oplus 1)$ formula $\phi = \forall x \exists y \leqslant x.\chi$ with χ quantifier-free, an equivalent WCMA can be constructed.*

Proof. Again, following [16] we can write χ in disjunctive normal form

$$\bigvee_i \big(\alpha_i(x) \wedge \beta_i(y) \wedge \delta(x,y) \wedge \epsilon(x,y)\big)$$

where α_i and β_i are as before, $\delta_i(x,y)$ is either $x = y \oplus 1$ or $x \neq y \oplus 1$, and $\epsilon_i(x,y)$ is either $x = y + 1$ or $x \neq y + 1$. At this point it is possible to directly construct a WCMA, simultaneously checking for satisfaction of any of the disjunctive clauses for each position in the word. This can be done by storing in the automaton state, as it goes along, whether 0, 1, or at least 2 satisfying positions have already been found, and this information in conjunction with the class memory function and knowledge of the S_{+1} predicate is sufficient to check that each position statisfies χ as it reads that position. □

3.4 Summary

In this chapter we introduced weak class memory automata, in which the local acceptance condition of class memory automata is dropped. We showed that the emptiness problem for these automata is ExpSpace-complete, and that deterministic WCMA are closed under all Boolean operations (in PTime). A summary of the closure properties of CMA and their deterministic and weak variants is shown in Table 3.1.

Table 3.1: Closure properties of CMA

CMA class	$\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$	$\mathcal{L}(\mathcal{A}) \cup \mathcal{L}(\mathcal{B})$	$\overline{\mathcal{L}(\mathcal{A})}$	$\mathcal{L}(\mathcal{A}) \cdot \mathcal{L}(\mathcal{B})$	$\mathcal{L}(\mathcal{A})^*$
CMA	✓	✓	X	✓	X
Det. CMA	✓	X	X	X	X
Weak CMA	✓	✓	X	✓	X
Det. Weak CMA	✓	✓	✓	X	X

We then showed that WCMA are equivalent to three existing forms of automata: locally prefix-closed data automata, class counting automata, and non-reset history register automata. Finally, we identified a new fragment of existential monadic second-order logic, which we called the *existentially bounded* fragment, and showed that this fragment is equiexpressive with WCMA.

In the next chapter we will see how this weakness restriction allows us to recover decidability from a form of class memory automata that otherwise have an undecidable emptiness problem.

Chapter 4

Class Memory Automata over Nested Data

In Chapter 2 we motivated data words as useful in, among other areas, modelling concurrent systems: each process can be identified by a data value, and class memory automata can then verify properties of the system. What happens when these processes can spawn subprocesses, which themselves can spawn subprocesses, and so on? In these situations the parent-child relationship between processes becomes important, and a single layer of data values cannot capture this; instead we want a notion of *nested* data values, which themselves contain the parent-child relationship. Such nested data values have applications beyond just in concurrent systems: they are prime candidates for modelling many computational situations in which names are used hierarchically. This includes higher-order computation where intermediate functional values are being created and named, and later used by referring to these names.

In this chapter we begin by recapping a presentation of nested data values in [13], and introduce another presentation, more closely mirroring the idea of using nested data values as names for subprocesses of subprocesses. We then define a natural extension of class memory automata to these nested data values. We call these automata *nested data class memory automata* (NDCMA). In Section 4.2 we begin looking at the basic algorithmic properties of these NDCMA. We find that emptiness for these automata is undecidable in general, but becomes decidable when the *weakness* condition discussed in Chapter 3 is reintroduced. We then examine links between NDCMA and existing models of computation. We show that NDCMA

are equivalent to the nested data automata (NDA) introduced in [22], with weakness corresponding to – as in the normal CMA case – local prefix-closure.

This chapter presents material and results first published in [19]. It is joint work with Andrzej Murawski and Luke Ong.

4.1 Nested Data Values & Class Memory Automata

Nested data values were first introduced in [13]. These gave each letter in $\mathbb{D}$ a fixed number, k, of data values as an ordered tuple. Two letters, $(a, d_1, d_2, \ldots, d_k)$ and $(b, e_1, e_2, \ldots, e_k)$ are then equivalent up to level i if $d_1 = e_1$, $d_2 = e_2$, $\ldots$, and $d_i = e_i$. Data were then formed from these letters in the normal way.

Note that this setting implies that two letters agreeing on the ith data value has no significance if they disagree on the jth data value where $j < i$.

This definition means that every data value in the word has the same number of data values – which may be less representative if we are using data values as process and subprocess identifiers, which have a natural hierarchy. We therefore, in [19], introduced an alternative presentation of nested data values, allowing for different letters to have different levels of data values. In this presentation, rather than each letter holding the information to check for equivalence up to level i, we equip the underlying data set with a tree structure, and each letter may then just take one data value as before.

Definition 4.1.1. A *rooted tree* (henceforth, just *tree*) is a simple directed graph $\langle D, \mathit{pred} \rangle$, where $\mathit{pred} : D \rightharpoonup D$ is the predecessor map defined on every node of the tree except the root, such that every node has a unique path to the root. A node n of a tree has *level* l just if $\mathit{pred}^{l-1}(n)$ is the root (thus the root has level 1). A tree has *bounded level* just if there exists a least $l \geq 1$ such that every node has level no more than l; we say that such a tree has level l.

We define a *nested dataset* of level l, $\langle \mathcal{D}, \mathit{pred} \rangle$, to be a forest of infinitely many trees of level l which is *full* in the sense that for each data value d of level less than l, d has infinitely many children.

We note that the *pred* relation gives rise to a notion of automorphism on $\langle \mathcal{D}, \mathit{pred} \rangle$: those bijections of $\mathcal{D}$ that preserve the *pred*-structure. Generally we will not be

concerned with the individual members of $\mathcal{D}$ used, but their position in the *pred*-structure, and hence the languages we define over these nested datasets will be closed under these automorphisms.

Example 4.1.1. We consider a simple concurrent system of (an unbounded number of) top-level processes that can be in one of two states, a or b, and sub-processes that, once spawned, can only be in one state, c. We stipulate that a top-level process can only spawn sub-processes if in state a, but permit all other behaviour.

We can represent runs of this system as a data language using a nested dataset of level 2. Each top-level process will be represented by a level-1 data value, d, and its sub-processes will be identified by level-2 data values, d_i, such that $pred(d_i) = d$.

A word representing a run of a single one of these top-level processes is then, for a fixed level-1 data value d, generated by:

$$w ::= \epsilon \mid w \cdot (b, d) \mid w \cdot (a, d)(c, d')^* \text{ where each } d' \text{ appears only once, and } pred(d') = d$$

We denote the set of all words of this form L_d.

The language of possible runs of the whole system is then an interleaving of an unbounded number of words $w_d \in L_d$ for distinct level-1 data values d.

We note that these two presentations, which we refer to as the *nested* presentation and *tree* presentation respectively, have natural translations between them[1].

A word in the nested presentation is equivalent to a word in the tree presentation in which every letter's data value is of the lowest level. Agreement up to level i in the nested presentation is transformed into the two data values sharing the same level i ancestor. Formally, given a word in the nested presentation $w \in \Sigma \times \mathcal{D}^k$ we construct a nested dataset $\langle D, pred \rangle$ where D is the set of sequences in $\mathcal{D}^*$ of length at most k, and if $s \in \mathcal{D}^*$ and $d \in \mathcal{D}$, $pred(s\, d) = s$. Each letter $(a, d_1, \ldots, d_k)$ in w is then represented in the tree presentation by the letter $(a, (d_1 \ldots d_k))$.

Conversely, a word in the tree presentation can be deemed equivalent to a word of the nested presentation by using the finite alphabet part of the letter to specify the level of the data value, and dummy data values for the lower levels.

[1] We note that these translations do not give an isomorphism, as one of the main reasons for using the tree presentation is to be able to have letters with different levels of data values, which the nested presentation is unable to represent except on an ad-hoc basis

We now extend CMA to nested data by allowing the nested data class memory automaton (NDCMA), on reading a data value d, to access the class memory function's memory of not only d, but each ancestor of d in the nested data set. Once a transition has been made, the class memory function updates the remembered state not only of d, but also of each of its ancestors. Formally:

Definition 4.1.2 (Tree Presentation)**.** Fix a nested data set of level l. A *Nested Data CMA of level* l over the alphabet $\Sigma \times \langle \mathcal{D}, \mathit{pred} \rangle$ is a tuple $\langle Q, \Sigma, \delta, q_0, F_L, F_G \rangle$ 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, and δ is the transition map. δ is given by a union $\delta = \bigcup_{1 \leq i \leq l} \delta_i$ where each δ_i is a function:

$$\delta_i : Q \times \Sigma \times (Q_\perp)^i \to \mathcal{P}(Q)$$

The automaton is *deterministic* if each set in the image of δ is a singleton; and is *weak* if $F_L = Q$. A configuration is a pair (q, f) where $q \in Q$, and $f : \mathcal{D} \to Q_\perp$ is a class memory function (i.e. $f(d) = \perp$ for all but finitely many $d \in \mathcal{D}$). The initial configuration is (q_0, f_0) where f_0 is the class memory function mapping every data value to $\perp$. A configuration (q, f) is final if $q \in F_G$ and $f(d) \in F_L \cup \{\perp\}$ for all $d \in \mathcal{D}$. The automaton can transition from configuration (q, f) to configuration (q', f') on reading input (a, d) just if d is a level-i data value, $q' \in \delta(q, a, (f(\mathit{pred}^{i-1}(d)), \ldots, f(\mathit{pred}(d)), f(d)))$, and $f' = f[d \mapsto q, \mathit{pred}(d) \mapsto q, \ldots, \mathit{pred}^{i-1}(d) \mapsto q]$. A run $(q_0, f_0), (q_1, f_1), \ldots, (q_n, f_n)$ is accepting if the configuration (q_n, f_n) is final. $w \in L(\mathcal{A})$ if there is an accepting run of $\mathcal{A}$ on w.

Again, we call the $(Q_\perp)^i$ part of the transition function input the *signature* of the transition, and we will generally for clarity write the signature as a column vector. Similar to the CMA case, we may write $q' \in \delta(q, a, \bar{s})$ as $q \xrightarrow{a, \bar{s}} q'$.

Example 4.1.2. We return to the language in Example 4.1.1, and give a (weak deterministic) NDCMA that recognises the language described there.

Since the language recognised is just an interleaving of the language for each data value, the current state of the automaton will not affect the transitions available. Instead, the current state will store whether the level-1 data value was last seen reading a or b (i.e. whether or not that top-level process can currently spawn subprocesses).

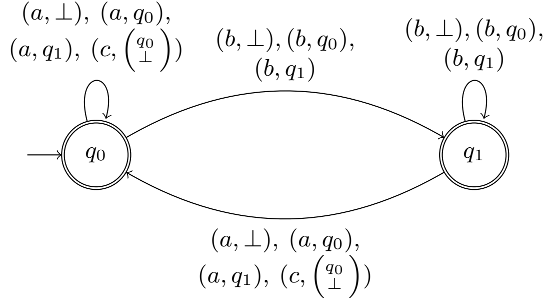


Figure 4.1: An example weak deterministic level-2 NDCMA

We use states q_0 and q_1 respectively for this. Reading a level-2 data value is then only possible when its parent data value was last seen in q_0 - this is encoded by having available transitions when reading input c with data signature $\binom{q_0}{\perp}$, but not permitting transitions with data signature $\binom{q_1}{\perp}$.

The automaton is depicted in Fig. 4.1.

We also present a version of these automata over the style of nested data values used in [13]:

Definition 4.1.3 (Nested Presentation)**.** A *Nested Data CMA of level k* over the alphabet $\Sigma \times \mathcal{D}^k$ is a tuple $\langle Q, \Sigma, \delta, q_0, F_L, F_G \rangle$ where Q is a finite set of states, $q_0 \in Q$, $F_G \subseteq F_L \subseteq Q$, and $\delta : Q \times \Sigma \times (Q_\perp)^k \to \mathcal{P}(Q)$ is the transition map.

A configuration is a tuple $(q, f_1, f_2, \ldots, f_k)$, where each $f_i : \mathcal{D}^i \to Q_\perp$ maps an i-tuple of data values to a state in the automaton (or $\perp$). The initial configuration is $(q_0, f_1^0, \ldots, f_k^0)$ where f_i^0 maps every tuple in the domain to $\perp$. A configuration $(q, f_1, \ldots, f_k)$ is final if each f_i maps into $F_L \cup \{\perp\}$. The automaton can transition from configuration $(q, f_1, \ldots, f_k)$ to configuration $(q', f'_1, \ldots, f'_k)$ on reading input $(a, d_1, \ldots, d_k)$ just if $q' \in \delta(q, a, (f_1(d_1), f_2(d_1, d_2), \ldots, f_k(d_1, \ldots, d_k)))$, and each $f'_i = f_i[(d_1, \ldots, d_i) \mapsto q']$.

It is straightforward to see how nested-presentation NDCMA can be seen as tree-presentation NDCMA. A little more care must be taken when transforming the tree-presentation into nested-presentation, but this is also not difficult:

Example 4.1.3. We show how the automaton in Example 4.1.2 can be transformed into a nested-presentation NDCMA.

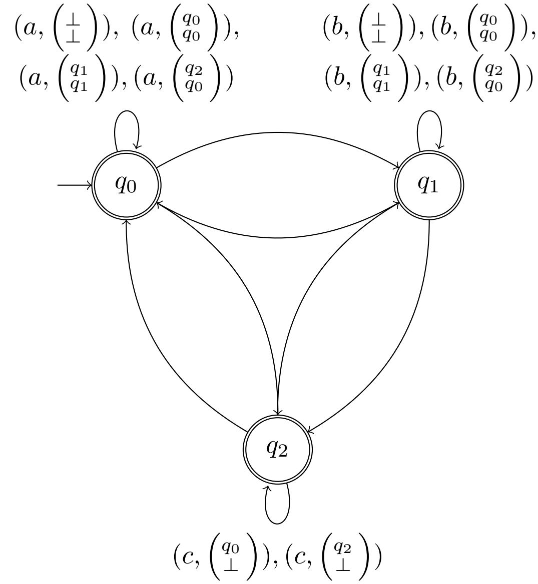


Figure 4.2: A nested-presentation weak deterministic level-2 NDCMA

The difficulty lies in the fact that, unlike in the tree-presentation case, all letters will have two layers of data values. Hence, we must use "dummy" data values for the second layer when we only have one meaningful layer. We can achieve this by, when a level-1 data value is created, simultaneously creating a level-2 under it, and using that level-2 value whenever we only mean to use the top-level of data values. This means we will have to keep this level-2 value separate from the level-2 values that we use to identify sub-threads of the top-level moves. We can do this by having them read in different states. We therefore introduce an additional state to the automaton in Example 4.1.2, which we move to exclusively when reading a c-subthread.

The automaton is depicted in Fig. 4.2. For the same reasons as before, in this automaton the current state does not affect the transitions, and so we only label the transitions available moving to each state once.

4.2 Nested Data CMA Basic Properties

Having defined a natural notion of CMA of nested data values, we investigate their algorithmic properties.

It is straightforward to show that the constructions for CMA and their variants for intersection, union, complementation, and concatenation that were summarised

in Table 3.1 all descend to NDCMA. Further, since level-1 NDCMA are equivalent to CMA, it is immediate that NDCMA cannot have "better" closure properties than CMA. Hence the closure properties for (weak) (deterministic) NDCMA are the same as for (resp. weak) (resp. deterministic) CMA.

We now ask whether emptiness is decidable. For CMA, it was seen that emptiness was equivalent to VASS reachability, and for weak CMA this reduced to coverability. By small alterations to this argument, we can show that reset VASS reachability (resp. coverability) can be reduced to emptiness of NDCMA (resp. weak NDCMA). Thus we get the following result:

Theorem 4.2.1. *Strong (resp. weak) nested data class memory automata have an undecidable (resp. Ackermann-hard) emptiness problem.*

Proof. As mentioned, this proof is an adaptation of the reduction from VASS to CMA emptiness presented in Section 2.3.1. In particular, a second layer of data values allows the "increments" to be 'grouped', and then all of the increments under a single data value discard simultaneously - the behaviour of reset VASS. See Section 2.1 for a summary of rVASS.

Given a rVASS $(Q, k, \Delta_i \cup \Delta_d \cup \Delta_0)$ we construct an NDCMA with states $(\{1, \ldots, k\} \times (Q \uplus \bar{Q} \uplus \{*\})) \uplus \Delta_0$, where $\bar{Q}$ is a second copy of Q, and an additional (initial) state $*$. The states $\{*\} \cup (\{1, \ldots, k\} \times \{*\})$ will be used just in setting up the automaton. Being in a state (i, q) or $(i, \bar{q})$ where $q \in Q$ will correspond to the rVASS being in state q, whilst being in state $\delta \in \Delta_0$ will correspond to being part-way through executing the transition δ. The $1, \ldots, k$ part of the state will be stored in the CMF to remember which counter the data value corresponds to. Only one level-1 data value for each counter will be "active" at a time. Increments for a counter will then be level-2 data values nested under the level-1 data value for that counter, and the CMF will remember them as being in a state in $\{1, \ldots, k\} \times Q$. When decremented they will then be moved to a state in $\{1, \ldots, k\} \times \bar{Q}$, from where they cannot be used in a decrement again. A reset transition $\delta = (q, 0_i, q')$ will be modelled by first moving to the state δ reading the level-1 data corresponding to counter i, which is being reset, then reading a fresh level-1 data value and going to state $(i, \bar{q'})$, thus setting up the new level-1 data value for that counter. Since δ-states will not occur in the signature

of any transitions, this is sufficient to prevent level-2 data values nested under this level-1 data value be used in future decrements.

Since we are just concerned with emptiness of the automaton, we ignore the Σ-part of the transitions. We now describe the transition relation. Note that for $j \in \mathbb{N}$ and $q \in Q$ the outgoing transitions from state (j, q) will be the same as the outgoing transitions from $(j, \bar{q})$, hence we only describe the transitions from (j, q).

- there is a transition $* \xrightarrow{\perp} (1, *)$;
- for $1 \leq i \leq k-2$ there is a transition $(i, *) \xrightarrow{\perp} (i+1, *)$;
- there is a transition $(k-1, *) \xrightarrow{\perp} (k, \bar{q_0})$;
- for each increment transition $(q, \bar{e}_i, q') \in \Delta_i$ there are transitions $(j, q) \xrightarrow{\binom{(i,p)}{\perp}} (i, q')$ where $j \in \{1, \ldots, k\}$ and $p \in Q \cup \bar{Q} \cup \{*\}$;
- for each decrement transition $(q, -\bar{e}_i, q') \in \Delta_d$ there are transitions $(j, q) \xrightarrow{\binom{(i,p)}{(i,p')}} (i, \bar{q'})$ where $j \in \{1, \ldots, k\}$, $p \in Q \cup \bar{Q}$, and $p' \in Q$;
- for each reset transition $\delta = (q, 0_i, q') \in \Delta_0$ there are transitions $(j, q) \xrightarrow{(i,s)} \delta$ where $s \in Q \cup \bar{Q} \cup \{*\}$, and $\delta \xrightarrow{\perp} (i, \bar{q'})$. There are also transitions $\delta \xrightarrow{\binom{\delta}{(i,p)}} \delta$ where $p \in Q$, to allow prior increments to be moved to a locally accepting state.
- for each of the transitions added above from state (j, q), where $j \in \{1, \ldots k\}, q \in Q$, there is an identical transition originating from $(j, \bar{q})$.

The globally accepting states are then $\{1, \ldots, k\} \times (Q \cup \bar{Q})$. The locally accepting states are $\{1, \ldots, k\} \times (\bar{Q}) \cup \{*\}) \cup \Delta_0$. This leaves just the states $\{1, \ldots, k\} \times Q$ as non-locally accepting, since these are the states visited only immediately after increment transitions, and so if these increments are decremented or reset, there will be the opportunity for the automaton to move the data values out of these states. $\square$

Having given a non-primitive recursive lower-bound for weak NDCMA emptiness, we now show that the problem is in fact decidable. We do this by reduction to a well-structured transition system (WSTS) – first introduced in [29] – a broad framework

subsuming VASS and rVASS, and a wider set of formalisms including process algebras and string rewrite systems [30]. We give a brief introduction to WSTS, and then prove the result.

Definition 4.2.1 ([30])**.** A *quasi-ordering* on a set X is a reflexive and transitive relation $\leq \subseteq X \times X$. A *well-quasi-ordering* (wqo) is a quasi-ordering such that any infinite sequence $x_0, x_1, \ldots$ has some $i < j$ such that $x_i \leq x_j$. Given a quasi-ordering, an *upward closed set* is a set I such that if $x \in I$ and $x \leq y$ then $y \in I$. For any $x \in X$ we can define an upward closed set $\uparrow x = \{y \mid x \leq y\}$. A *basis* of an upward closed set I is a set I^b such that $I = \bigcup_{x \in I^b} \uparrow x$.

A *well-structured transition system* is a triple $\langle S, \rightarrow, \leq \rangle$ where S is a set of states, $\rightarrow \subseteq S \times S$ is a transition relation on S, $\leq$ is a wqo on S, and the following additional conditions hold:

- (finite branching) given $s \in S$ the set of successor states of s, $\{t \in S \mid s \rightarrow t\}$, is finite; and
- (upward-compatibility) if $s_1 \leq t_1$ and $s_1 \rightarrow s_2$ then there exists a sequence of transitions $t_1 \rightarrow^* t_2$ such that $s_2 \leq t_2$.

For $s \in S$ we further define the set $\texttt{Pred}(s) = \{t \in S \,|\, t \rightarrow s\}$, and we say a WSTS has *effective pred-basis* if there is an algorithm taking any $s \in S$ and returning a finite set $pb(s)$ that is a finite basis of $\uparrow \texttt{Pred}(\uparrow s)$.

We then get the key result, which states that any WSTS and with effective pred-basis and decidable $\leq$ has a decidable covering problem [30]. Here the covering problem takes input states s and t and returns whether it is possible, starting from s, to reach a state covering t. That is, whether $\exists t' \geq t$ such that $s \rightarrow^* t'$.

We now prove the decidability of weak NDCMA emptiness:

Theorem 4.2.2. *Weak NDCMA emptiness is reducible to the covering problem for a WSTS.*

Proof. For this we will fix a nested data set, $\langle \mathcal{D}, pred \rangle$ of level k, and we make the following observation: a class memory function $f : \mathcal{D} \rightarrow Q_\perp$ can be interpreted as a labelling of the forest $\mathcal{D}$ with labels from $Q_\perp$. To simplify the following discussion, we add a distinguished "level-0" data value, which is the parent of all of the level-1 data

values; this means we are working only over a tree, rather than a forest. Further, we note that by the definition of the transition function, if f is a class memory function obtained as part of a reachable configuration (q, f), then $f(d) = p \in Q$ implies $f(pred(d)) \neq \perp$ (here we assume that the class memory function has some distinguished symbol for the level-0 data value). Hence, in our labelled tree $\mathcal{D}_f$, whenever we reach a node labelled $\perp$, all of that node's descendants are also labelled $\perp$. Thus, each reachable class memory function gives rise to an unordered labelled finite tree, T_f (with labels from Q plus a distinguished symbol for the root) of depth $\leqslant k+1$. Further, given two class memory functions f and f' such that $T_f = T_{f'}$, it is clear that f is equivalent to f' up to automorphism of $\mathcal{D}$ (i.e. renaming of data values).

Given an NDCMA $\mathcal{A} = \langle Q, \Sigma, \Delta, q_0, F \rangle$ we construct the WSTS $\mathcal{S} = \langle S, \rightarrow, \leq \rangle$ as follows:

- $S = Q \times \Phi$ where Φ is the set of unordered labelled finite trees of depth $\leqslant k+1$ (with labels from Q plus a distinguished symbol for the root)

- $\rightarrow$ is defined as follows: $(q, T) \rightarrow (q', T')$ iff there are Class Memory Functions f_T and $f_{T'}$ s.t. $T_{f_T} = T$ and $T_{f_{T'}} = T'$ and $(q', f_{T'})$ is a valid successor configuration (in $\mathcal{A}$) of (q, f_T).

- $\leq$ is defined as follows: $(q, T) \leq (q', T')$ iff $q = q'$ and there is a tree homomorphism from T to T'.[2]

That $\rightarrow$ is finite-branching is obvious. That $\leq$ is a qo is immediate from composition of homomorphisms. That it is a well-quasi-order is given by Lemma 7 in [59]. The upward-compatibility condition is straightforward: if $(p, R) \leq (q, T)$ and $(p, R) \rightarrow (p', R')$ then $p = q$ and there is some $\delta \in \Delta$ such that applying δ from configuration (p, f_R) yields configuration $(p, f_{R'})$. Applying the same δ from configuration (q, T) will yield an appropriate $(p', T') \geq (p', R')$.

Decidability of $\leq$ is also straightforward: to decide whether $(q, T) \leq (q', T')$ first check $q = q'$, then simply check each possible injection from T to T' for being a homomorphism.

[2]A tree homomorphism from T_1 to T_2 is a function, ϕ, mapping each node of T_1 to a node of T_2 which maps the root node to the root node, and preserves *pred* and labels. (i.e. $\phi(pred(n)) = pred(\phi(n))$ and $lab(n) = lab(\phi(n))$.)

The only remaining property to check is that we have an effective pred-basis. We do this by showing an effective pred-basis wrt each transition in $\mathcal{A}$. The union of these can then be taken. Formally, we compute $pb(s)$ as $pb(s) = \bigcup_{\delta \in \Delta} S_{s,\delta}$.

We now define these $S_{s,\delta}$. Assume $s = (q, T)$. If δ is a transition which does not go to state q, $S_{s,\delta} = \emptyset$. Otherwise, $\delta = p \xrightarrow{a,(i,\bar{s})} q$ for appropriate p, a, i, and $\bar{s}$. In this case $S_{s,\delta} = \{T_{\rho,\bar{s}} \,:\, \rho \in R_{T,\bar{s}}\}$, where

$R_{T,\bar{s}} =$ the set of downward-paths, $n_0, n_1, \ldots n_j$, from the root, such that the labels of this path (excluding the root's label) give some non-empty prefix of q^k and if $s_k = \perp$ then $n_k$has no children other than (possibly) n_{k+1}

$T_{\rho,\bar{s}}$ is then defined as follows:

- If ρ is a path of i nodes (i.e. the path labels were q^i), then $T_{\rho,\bar{s}}$ is constructed by replacing the labels of the nodes in ρ with the corresponding element of $\bar{s}$. If such an element is $\perp$ then the node is deleted (note that in this case by choice of ρ any child nodes must also be in ρ and labelled with $\perp$, so are also deleted)[3]
- If ρ is a path of $1 \leqslant j < i$ nodes, then $T_{\rho,\bar{S}}$ is constructed by replacing the labels of the nodes in ρ with the corresponding element of $\bar{s}$, and adding nodes labelled with the remaining elements of $\bar{s}$ as a branch off the last node in ρ. Any nodes labelled with $\perp$ are then deleted (again, due to the choice of ρ and constraints on transitions this must result in obtaining a tree).

□

4.3 Link with Nested Data Automata

In [22] Decker et al. also examined "Nested Data Automata" (NDA), and showed the locally prefix-closed NDA (pNDA) to have decidable emptiness (via reduction to well-structured transition systems). In fact, these NDA precisely correspond to NDCMA, and again being locally prefix-closed corresponds to weakness. In this section we briefly outline this connection.

[3]Note that the union of the $T_{\rho,\bar{s}}$ for these paths actually give $Pred((q, T))$. The upward closure is obtained from the shorter paths in the next case.

Definition 4.3.1. A k-nested data automaton (NDA) is a tuple $(\mathcal{A}, \mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_k)$ where $(\mathcal{A}, \mathcal{B}_i)$ is a data automaton for each i. Such automata run on words over the alphabet $\Sigma \times \mathcal{D}^k$, where $\mathcal{D}$ is a (normal, unstructured) dataset. As for normal data automata, the transducer, $\mathcal{A}$, runs on the string projection of the word, giving output w. Then for each i the class automaton $\mathcal{B}_i$ runs on each subsequence of w corresponding to the positions which agree on the first i data values. The NDA is locally prefix-closed if each $(\mathcal{A}, \mathcal{B}_i)$ is.

By amending the proof that CMA and DA are equivalent, we can show a similar equivalence between NDCMA and NDA. As in Chapter 3, weakness of the NDCMA will correspond to prefix-closure of the class automata of the NDA.

Theorem 4.3.1. *Nested data class memory automata (nested presentation) are equivalent to nested data automata, with weakness corresponding to local prefix-closure.*

Proof. This is a straightforward adaptation of the proof of Proposition 2.3.1 - the equivalent result in the non-nested setting. We provide the constructions for completeness.

Reducing NDA to NDCMA. We assume we are given a NDA $(\mathcal{A}, \mathcal{B}_1, \ldots, \mathcal{B}_k)$ where $\mathcal{A}$ has set of states $Q_{\mathcal{A}}$ and transition relation $\Delta_{\mathcal{A}}$, and each $\mathcal{B}_i$ has set of states Q_i and transition relation Δ_i. We further assume $\mathcal{A}$ has initial state q_0 and final states $F_{\mathcal{A}}$, and that $\mathcal{B}_i$ has initial state q_0^i and final states F_i.

We construct a NDCMA $\mathcal{C}$ with states $Q_{\mathcal{A}} \times Q_1 \times \cdots \times Q_k$. We write these states as $(s, \bar{s})$ where $s \in Q_{\mathcal{A}}$ and $\bar{s}$ is the vector of states over $Q_1 \times \cdots \times Q_k$. The automaton being in state $(q, q_1, \ldots, q_k)$ corresponds to the base automaton being in state q, and the ith class automaton being in state q_i.

For each transition $q \xrightarrow{a} (q', \alpha)$ in $\Delta_{\mathcal{A}}$, $p_1 \xrightarrow{\alpha} p_1'$ in Δ_1, ..., and $p_k \xrightarrow{\alpha} p_k'$ in Δ_k, we have transitions in $\mathcal{C}$ of the form $(q, \bar{s}) \xrightarrow{(a,\bar{p})} (q', \bar{p'})$ where $\bar{s}$ is any element of $Q_1 \times \cdots \times Q_k$, $\bar{p} = (p_1, \ldots, p_k)$, and $\bar{p'} = (p_1', \ldots, p_k')$. We also have, where p_j, ..., p_k above are q_j, ..., q_k respectively, the transitions $(q, \bar{s}) \xrightarrow{(a,\bar{r})} (q', \bar{p'})$ where $\bar{s}$ and $\bar{p'}$ are as before, and $\bar{r} = (p_1, \ldots, p_{j-1}, \perp, \ldots, \perp)$.

The locally accepting states are $Q \times F_1 \times \cdots \times F_k$. The globally accepting states are $F_{\mathcal{A}} \times F_1 \times \cdots \times F_k$. Note that if the NDA is locally prefix closed, then every state of $\mathcal{C}$ is locally accepting, and so $\mathcal{C}$ is weak.

Reducing NDCMA to NDA. Again, this will work by the base automaton attempting to simulate the NDCMA by itself, guessing the CMF, and the class automata then checking that the CMF values were correctly guessed.

Given a NDCMA $\mathcal{C} = \langle Q, \Sigma, \Delta, q_0, F_L, F_G \rangle$ we construct a NDA with base automaton $\mathcal{A}$ with state set Q and output alphabet $(Q_\perp)^k$. For each transition $q \xrightarrow{a,\bar{s}} q'$ in Δ there is a transition $q \xrightarrow{a} q'$ outputting $(\bar{s}, q')$. The ith class automaton then checks that the first letter in its input $(bars, q')$ has $\perp$ in the ith place of $\bar{s}$, and subsequently that the ith place of the first component matches the second component of the previous letter. It accepts if the second component of the final letter is a locally accepting state. □

4.4 Summary

We have introduced nested data class memory automata, and shown that while these automata have undecidable emptiness problem in general, by reintroducing the weakness constraint from Chapter 3 we recover decidability.

In [14], CMA were introduced as a simpler, equivalent, alternative to data automata. Similarly, we find NDCMA are equivalent to nested data automata from [22] – with weakness corresponding to local prefix-closure. Crucially, the fact that NDCMA – unlike NDA – have a deterministic variant gives us a subclass of NDCMA with a decidable equivalence problem, namely weak deterministic NDCMA. We will make use of this subclass in Chapter 7, where decidability of equivalence will be key in our results.

Chapter 5

Visibly Pushdown Class Memory Automata

We are interested in augmenting class memory automata with a stack. In this chapter we introduce a notion of *visibly pushdown class memory automata* (VPCMA), which naturally combine the class memory aspects of CMA with the visible pushdown aspects of VPA, but also add data values to the stack so that corresponding push- and pop-moves must share the same data value. In addition, we introduce a slight variant of VPCMA with a run-time constraint on the words accepted. 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 expressively different, we show these two models have equivalent emptiness problems.

Further, we show that the emptiness problem of VPCMA is equivalent to the reachability problem for extended branching VASS (EBVASS) [45]. Whether or not EBVASS reachability is decidable is an open problem, which we do not address here. Unlike in CMA and NDCMA cases, we find that weakness does not affect the hardness of the emptiness problem for VPCMA, as the stack can be used to check the local acceptance condition. However, weakness does, again, help with the closure properties of the languages recognised.

We briefly sketch the structure of this chapter. In Section 5.1 we define VPCMA, show that weakness does not change the difficulty of emptiness, and present basic closure properties. In Section 5.2 we introduce EBVASS, and prove that VPCMA emptiness is reducible to EBVASS reachability. In Section 5.3 we introduce the run-time

variant mentioned, and call the resulting automata *scoping* VPCMA (SVPCMA). We again show basic closure properties, as well as showing that their emptiness problem is equivalent to that for VPCMA. Finally, in Section 5.4 we demonstrate how EBVASS reachability can be reduced to emptiness of scoping VPCMA, thus completing the proof of equivalence between EBVASS reachability and VPCMA emptiness.

5.1 Definition and Basic Properties

We define visibly pushdown class memory automata in a fairly natural way, combining the key aspects of both visibly pushdown automata and class memory automata. As with VPA, these automata will have a stack, and whether a push, pop, or noop-move is carried out will be determined by the input letter. As with CMA, our VPCMA will have a class memory function that, for each data value seen in the run, will remember the state that data value was last seen in. The only subtlety in how these two ideas are combined is in the contents of the stack: we allow each stack element to be a pair combining a letter from a finite alphabet with the data value that was read when this element was pushed to the stack. 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. Formally:

Definition 5.1.1. Recall from Section 2.2 a *pushdown alphabet* is a triple $(\Sigma_{\texttt{push}}, \Sigma_{\texttt{pop}}, \Sigma_{\texttt{noop}})$, where $\Sigma_{\texttt{push}}$, $\Sigma_{\texttt{pop}}$, and $\Sigma_{\texttt{noop}}$ are disjoint finite sets of symbols.

Fix a dataset $\mathcal{D}$. A *visibly pushdown class memory automaton* (VPCMA) 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, $\Sigma = (\Sigma_{\texttt{push}}, \Sigma_{\texttt{pop}}, \Sigma_{\texttt{noop}})$ is a pushdown alphabet, and Γ a finite stack alphabet. Δ is the transition relation where:

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

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) whilst in configuration (q, f, S) the automaton can follow transitions as follows:

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

Acceptance of words is then defined in the normal way, with a word being accepted just if there is a run of the word from the initial configuration to an accepting configuration.

Determinism is defined in the normal way. That is, a VPCMA is deterministic just if the conditions:

(i) $(q, s, a_{\texttt{push}}, \gamma, p), (q, s, a_{\texttt{push}}, \gamma', p') \in \Delta \;\Rightarrow\; \gamma = \gamma', p = p'$;

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

(iii) $(q, s, a_{\texttt{noop}}, p), (q, s, a_{\texttt{noop}}, p') \in \Delta \;\Rightarrow\; p = p'$

all hold.

We give a simple example of a language recognisable by VPCMA but not by normal CMA.

Example 5.1.1. Consider the language consisting of words generated by:

$$w ::= \epsilon \;\mid\; (a, d) \cdot w \cdot (b, d) \quad d \notin w.$$

That is, the language will have string projection $\{a^n b^n : n \in \mathbb{N}\}$, each data value will occur once as an a-position and once as a b-position, and the order in which the data values occur as b-positions is the reverse of the order in which they occur as a positions.

We give a VPCMA recognising this language in Fig. 5.1, in which $\Sigma_{\texttt{push}} = \{a\}$ and $\Sigma_{\texttt{pop}} = \{b\}$. Note that since for the VPCMA to accept the stack must be empty, we do not need to use the local acceptance condition in recognising this language.

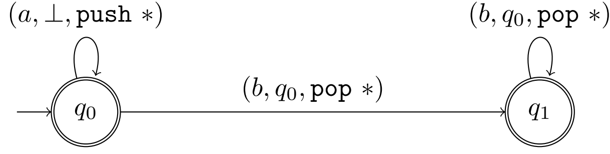


Figure 5.1: An example VPCMA

5.1.1 Weak VPCMA

Note that in the above example, we do not need to use the local acceptance condition, as the stack lets us ensure that each data value is revisited. This idea can be used to show that – unlike in the normal CMA case – the emptiness problem for weak VPCMA is no easier than that for VPCMA.

Definition 5.1.2. As for normal CMA, a VPCMA is weak if the set of locally accepting states is the whole set of states.

Proposition 5.1.1. *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 $\mathcal{A} = \langle\, Q, (\Sigma_{\texttt{push}}, \Sigma_{\texttt{pop}}, \Sigma_{\texttt{noop}}), \Gamma, \Delta, q_0, F_G, F_L \,\rangle$ is a VPCMA. We construct a weak VPCMA

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

where Δ' is given as follows:

- From the state `start` we have the push-transitions allowing new data values: $(\texttt{start}, \bot, \downarrow, *, \texttt{start}) \in \Delta'$ and $(\texttt{start}, \bot, \downarrow, *, q_0) \in \Delta'$.
- For each transition $t \in \Delta$ we have the transition $t[\bot \mapsto \texttt{start}] \in \Delta'$.

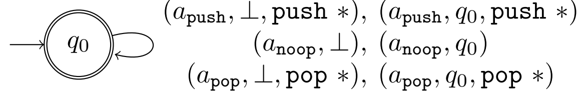


Figure 5.2: A "universal" VPCMA

- For each transition $(q, ..., q') \in \Delta$ where $q' \in F_G$, we have the transition $(q, \ldots, \texttt{end}) \in \Delta'$.
- Finally, for each $q \in F_L$ we have the pop-transition $(\texttt{end}, q, \uparrow, *, \texttt{end})$.

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

5.1.2 Closure Properties

VPCMA over a pushdown alphabet $(\Sigma_{\texttt{push}}, \Sigma_{\texttt{pop}}, \Sigma_{\texttt{noop}})$ take as input words in $\mathcal{D} \times (\Sigma_{\texttt{push}} \cup \Sigma_{\texttt{pop}} \cup \Sigma_{\texttt{noop}})$. However, they can only accept words that are "well-bracketed" by push and pop moves. That is, words that may be generated by the following grammar:

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

where d ranges over $\mathcal{D}$, and $a_{\texttt{push}}$, $a_{\texttt{pop}}$, and $a_{\texttt{noop}}$ range over $\Sigma_{\texttt{push}}$, $\Sigma_{\texttt{pop}}$, and $\Sigma_{\texttt{noop}}$ respectively. Equivalently, words recognised by the universal VPCMA – the VPCMA that accepts every transition possible and for which all states are accepting – an example universal VPCMA is depicted in Fig. 5.2. Hence, if we wish to consider whether a VPCMA can be complemented, it will be with respect to words that can be generated by the above grammar/accepting by a universal VPCMA.

Note that VPCMA can trivially simulate normal CMA, by not using the stack, and so the possible effective constructions can be at best the same as for normal CMA. In fact, this is the case, as due to the visible nature of the stack the product constructions do work. Further, in the deterministic weak case, the same trick of reversing the final states will give the complement. The only slight subtlety is that in constructing an automaton to recognise the concatenation of two languages, as well as using disjoint copies of the states of the two constituent automata, the stack alphabet should consist of disjoint copies of the stack alphabets of the constituent automata. This effectively

enforces that all pushes made in the run of the first automaton are popped before moving to the second automaton. Hence, the closure properties of VPCMA and their restrictions are the same as for normal CMA, which are summarised in Table 3.1.

5.2 Reduction to EBVASS

In this section we review *extended branching VASS* (EBVASS), which were introduced in [45] and shown to be equivalent to a form of data tree automaton. EBVASS are slightly more powerful than branching VASS (BVASS), for which it is unknown whether the reachability is decidable. The purpose of this section is to show that emptiness of VPCMA can be reduced to the reachability problem for EBVASS.

5.2.1 Extended Branching VASS

We first present branching VASS. These are an extension of VASS, where as well as 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 transitioning to a pre-given state. These copies must then complete their runs independently. Formally:

Definition 5.2.1. A (top-down) *branching VASS* (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 for unary transitions to be able to only make a single increment or decrement, and for split transitions to make no increments or decrements. It is clear that this more powerful presentation does not change the power of the model, but it allows us a slightly more concise reduction from VPCMA.

We now move to give a definition of EBVASS. These were introduced in [45], and extend BVASS with the ability to split counters in more complex ways when a split transition is made. Although in [45] they were presented as symmetric, and bottom-up, we provide here a top-down asymmetric definition.

Definition 5.2.2. An *extended branching VASS* (EBVASS) is a tuple $(Q, q_0, L, k, \Delta_u, \Delta_s, C)$ such that $(Q, q_0, L, k, \Delta_u, \Delta_s)$ is a BVASS and $C \subseteq \{1, \ldots, k\}^3$ is the set of *constraints.*

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

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

- Suppose $C = \{c_1, \ldots, c_m\}$. If a node labelled $(q, \bar{v})$ has two child nodes then there is a transition $(q, \bar{u_1}, q', \bar{u_2}, q'', \bar{u_3}) \in \Delta_s$, $n_1, \ldots, n_m \in \mathbb{N}$, and $\bar{v_1}, \bar{v_2}, \bar{v_3} \in \mathbb{N}^k$ such that $\bar{v} = \bar{v_1} + \bar{u_1}$, $\bar{v_1} = \bar{v_2} + \bar{v_3} + \Sigma(n_i \cdot \bar{e}_{\pi_1(c_i)})$, and the left child node is labelled $(q', \bar{v_2} + \bar{u_2} + \Sigma(n_i \cdot \bar{e}_{\pi_2(c_i)}))$, and the right child node is labelled $(q'', \bar{v_3} + \bar{u_3} + \Sigma(n_i \cdot \bar{e}_{\pi_3(c_i)}))$.

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

5.2.2 VPCMA - EBVASS reduction

Theorem 5.2.1. *The emptiness problem for VPCMA is reducible to the reachability problem for EBVASS.*

5.2.2.1 Shape of the reduction

The central ideas of the reduction are as follows:

- 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 run of the VPCMA from state q to q'.

- 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.

To aid clarity of the proof, we will reduce weak VPCMA to EBVASS. By Proposition 5.1.1, this is sufficient.

5.2.2.2 Formal Reduction

Suppose $\mathcal{A} = \langle\, Q, \Sigma, \Gamma, \Delta, q_0, \{q_f\}\,\rangle$ is a weak VPCMA[1]. We construct an EBVASS $\mathcal{E}_{\mathcal{A}}$ such that its reachability problem is soluble 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 $e_{q,q'}$ for the vector in $\mathbb{Z}^k$ with a 1 in position $c_{q,q'}$ and 0 elsewhere. The set of constraints contains a constraint $c_{q,q''} \to c_{q,q'} \oplus c_{q',q''}$ 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 $a \in \Sigma_{\mathtt{noop}}$, we have:

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

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

$$(\textsc{push-pop}) \quad \frac{(q_1,p), \bar{v_1} + \bar{v_2} + e_{s,s''}}{(q_2,q_3), \bar{v_1} + e_{q_2,s'} \qquad (q_4,p), \bar{v_2} + e_{q_4,s''}}$$

  (Note that the above is a slight abuse of notation: split rules cannot also include increments[2]. 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.)

---

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

[2] 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

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

(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 $q \in Q$:

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

The rest of this subsection will be devoted to proving Theorem 5.2.1, by showing that $\mathcal{L}(\mathcal{A}) \neq \emptyset$ iff there is a run of $\mathcal{E}_{\mathcal{A}}$ reaching the target configurations. .

We first note this simple property of VPCMA:

Lemma 5.2.2. *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 5.2.3. *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 $f(d) \neq \bot$, $f'(d) = f(d)$.

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 $((q_1, q_2), \bar{v} + e_{\bot,q})$. 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 $f_1(d) = \bot = f_2(d)$. Then $\bar{v} + e_{\bot,q} \sim (f_1, f_2[d \mapsto q])$ 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:

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

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 Lemma 5.2.2 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 $(q_3, f_3', \gamma) \to_{\mathcal{A}} (q_4, f_4, \epsilon)$. 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 5.2.1: 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 5.2.4. *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 $a \in \Sigma_{\texttt{noop}}$, 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 5.2.1.

5.3 Scoping VPCMA

In Chapter 9 we will use VPCMA to encode a fragment of RML. In doing so we will use data values to represent names for processes, and pushes to the stack to represent starting a new (sub)thread of processes. In this case if a data value is first seen after a push-move, it would represent a process local to that thread. Hence, once the current stack element has been popped off the stack, it wouldn't make sense to see that data value again. To this end we introduce *scoping* VPCMA, which have precisely this restriction on reading data values.

In this section we will show that these scoping VPCMA have identical algorithmic properties to normal VPCMA, and that the two models can simulate one another.

5.3.1 Basic Properties

Definition 5.3.1. 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. 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.

Formally, this is achieved by having a configuration of a SVPCMA be a tuple (q, f, V, S) where q and f are states and CMFs as before, $V \subset_{fin} \mathcal{D}$ is the set of visible data values, and $S \in (\mathcal{D} \times \Gamma \times \mathcal{P}_{fin}(\mathcal{D}))^*$. The initial configuration is $(q_0, f_0, \emptyset, \epsilon)$, and a configuration is accepting just in the conditions set for normal VPCMA (i.e. no restrictions on V).

On reading an input letter (a, d) whilst in configuration (q, f, V, S), if $f(d) = \perp$ or $d \in V$ the automaton can follow transitions as follows:

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

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

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

Proposition 5.3.1. *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, that when popped checks it was in a locally accepting state, and prevents it from being used again. □

Similarly, all of the closure constructions that work for VPCMA work for SVPCMA (though this time closure is with respect to the set of languages generated by the universal SVPCMA).

5.3.2 Equivalence with VPCMA

It is clear that not all languages recognisable by VPCMA are recognisable by SVPCMA. It is also fairly straightforward to see that the language recognised by the universal SVPCMA on input alphabet ($\Sigma_{\texttt{push}} = \{a\}, \Sigma_{\texttt{pop}} = \{b\}, \Sigma_{\texttt{noop}} = \{c\}$) cannot be recognised by a VPCMA, since the VPCMA has no way of enforcing the scoping condition. Hence there cannot be effective translations between VPCMA and SVPCMA, but we now show that emptiness of one can be reduced to emptiness of the other. Hence their differences, whilst potentially useful, are essentially cosmetic.

To reduce emptiness of VPCMA to that of SVPCMA we employ a similar trick to that used when reducing 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 reduced 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. Thus we get:

Theorem 5.3.2. *Emptiness of VPCMA can be effectively reduced to emptiness of SVPCMA, and vice-versa.*

5.4 Reduction from EBVASS to SVPCMA

We now attempt to complete the picture for VPCMA by showing their emptiness problem is at least as hard as the reachability problem for EBVASS, and hence is decidable if and only if the EBVASS reachability problem is. We do this by reducing EBVASS reachability to emptiness of SVPCMA.

The key idea is that words over a pushdown alphabet can be viewed as trees by viewing them as their construction trees when generated by the grammar:

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

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

Class Memory Functions and finite labels. We noted in Section 2.3.1 that although class memory functions are normally of the form $f : \mathcal{D} \rightarrow Q_{\perp}$, it is straightforward to use an arbitrary finite set of labels instead of Q in this definition. We will take advantage of this in our reduction. Hence, our VPCMA will have a transition relation of the form

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

where $\mathtt{Lab}$ is our finite set of labels.

Simple EBVASS. As we discussed when we introduced BVASS in Section 5.2, we gave them the power to perform multiple increments and decrements in one transition. While this was useful in reducing VPCMA to EBVASS, we will now find it useful to

simply permit a single increment or decrement in unary transitions, and no increments or decrements in split transitions.

5.4.1 Reduction from BVASS

As dealing with constraints is the most complex part of the reduction, we will first simply reduce BVASS reachability to SVPCMA emptiness, and then extend this reduction to include constraints.

As when reducing VASS to CMA, in this reduction data values will be used to store the counter information. That is, the value of a counter will be represented by the number of data values that the class memory function assigns a label corresponding to that counter (we use the labels $1, \ldots, n$ for the n counters) . When a counter is incremented, a fresh data value is read, and given the appropriate label. When a counter is decremented, a data value with the label corresponding to that counter has its label changed to `done`. The fact that all increments have been decremented by the end of the run is then checked by the local acceptance condition.

Given a BVASS $\mathcal{B} = (Q, q_0, L, n, \Delta_u, \Delta_s)$, we construct a SVPCMA $\mathcal{A}_\mathcal{B}$ as follows:

- The set of states of $\mathcal{A}_\mathcal{B}$ is simply Q, the set of states of $\mathcal{B}$;
- The initial state is q_0, the initial state of $\mathcal{B}$;
- The set of labels is $\{1, \ldots, n\} \cup \{\texttt{done}, \texttt{split}\}$;
- The stack alphabet is Q;
- The pushdown alphabet is $(\Delta_s, \{\texttt{pop}\}, \Delta_u)$;
- The set of globally accepting states is L, the leaf states of $\mathcal{B}$;
- The set of locally accepting labels is $\{\texttt{done}\}$; and
- The transition relation of $\mathcal{A}_\mathcal{B}$ is defined as follows:
  - for each (unary) increment transition $\delta \in \Delta_u$ of the form $q \xrightarrow{+e_i} q'$ we have the transition $(q, \bot, \delta, q', i)$;
  - for each (unary) decrement transition $\delta \in \Delta_u$ of the form $q \xrightarrow{-e_i} q'$ we have the transition $(q, i, \delta, q', \texttt{done})$;

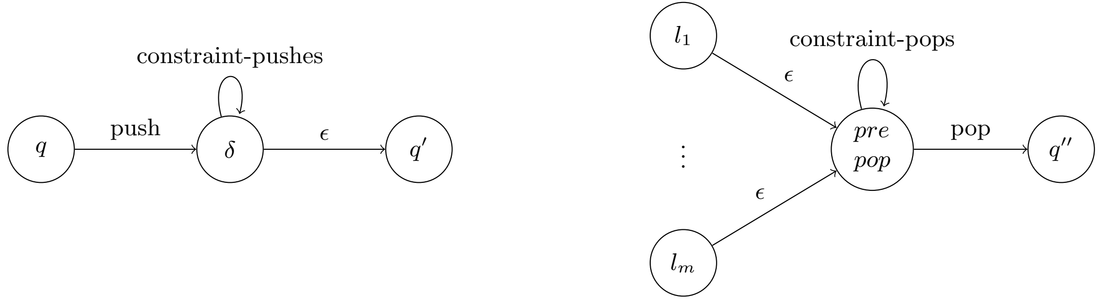


Figure 5.3: How an EBVASS split transition is modelled in an SVPCMA

- for each split transition $\delta \in \Delta_s$ of the form $q \to q' + q''$ we have the transition $(q, \bot, \delta, q'', q', \mathtt{split})$;
- finally, for each $l \in L$ and $q \in Q$ we have the transitions $(l, \mathtt{split}, \mathtt{pop}, q, q, \mathtt{done})$.

Proposition 5.4.1. *$\mathcal{L}(\mathcal{A}_\mathcal{B}) \neq \emptyset$ if and only if there is a run of $\mathcal{B}$ with root configuration $(q_0, \bar{0})$.*

Proof Sketch. The central idea is that each push- or noop-move in a run of $\mathcal{A}_\mathcal{B}$ corresponds to a split- or unary-transition in the corresponding run of $\mathcal{B}$. Pop-moves then correspond to finishing the left-hand branch of the split, and moving to the right-hand branch. With this correspondence in mind it is moderately straightforward to construct simulation proofs similar to those used in proving Theorem 5.2.1. □

5.4.2 Reduction from EBVASS

We now discuss how to extend the reduction from BVASS to SVPCMA to full EBVASS. That is, how constraints can be simulated.

The key idea here is to, after a push-transition corresponding to a split (i.e. those permitted in the BVASS reduction above), allow several more push-transitions corresponding to firings of the constraints. Firing a transition corresponding to a constraint (i, j, k) will take a data value with current label i, give it label j, and put a letter k on the stack. Then, when the corresponding pop-move is made, the label will be changed from *done* to k. The shape of the parts of the automaton corresponding to a split transition $\delta = (q, q', q'')$ is shown in Fig. 5.3. Transitions marked ϵ are silent-transitions that can be compressed out of the automaton.

There is a slight subtlety in the above, which is that in an EBVASS all constraints are fired simultaneously at a split transition, not sequentially. Hence we should be

sure that the same data value cannot be used to fire two constraints at the same split. Fortunately, this is already prevented, as if two such constraints were fired, when it came to make the corresponding pop-transitions, the first would fire correctly, but then the second could not because the data value would not have the `done`-label.

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

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

It should be clear that this alteration to the BVASS reduction deals with the constraints of extended EBVASS, and so we get the following result:

Theorem 5.4.2. *The reachability problem for EBVASS is reducible to the emptiness problem for SVPCMA.*

This completes the proof that emptiness problem for VPCMA and the reachability problem for EBVASS are equally hard. As it is unknown whether reachability for EBVASS (and indeed for BVASS) is decidable or not, the same is thus true for emptiness of VPCMA.

5.5 Summary

In this chapter we introduced visibly pushdown class memory automata, and scoping visibly pushdown class memory automata. These two models are natural extensions of class memory automata to include a stack and follow a visibly-pushdown discipline. The key point in how these automata are defined is that for words recognised by these automata, the matching push- and pop-inputs must share the same data value.

We showed that these two models, though different in expressive power, are essentially equally powerful as emptiness of one is reducible to emptiness of the other. Further, we showed that the stack can be used to enforce the local acceptance condition of class memory automata, and so the weak and strong variants of these automata are equally powerful. The closure properties of these automata were also shown to be the same as for normal class memory automata.

Finally and most importantly, we showed that these automata's emptiness problem is equivalent to the reachability problem for extended branching VASS, the decidability of which remains an open problem. In particular, reachability in EBVASS is a harder problem than reachability in BVASS, which is a long-standing open problem [21], though known to be at least 2-ExpSpace-hard [53]. We will use the automata we defined in this chapter later, in Chapter 9, in reductions to and from a fragment of RML.

Part II

Observational Equivalence of RML

Chapter 6

Preliminaries

In this chapter we introduce the background definitions, results, and methods on which our results in Part II will rely. In particular, we recap the definition of the programming language RML and its game semantic model. We also summarise previous decidability results for RML and related languages, and focus on how algorithmic game semantics has led to some of these results.

6.1 RML

RML is a call-by-value functional language with state [6]. It is similar to Reduced ML [79], the restriction of Standard ML to ground-type references, but is augmented with a "bad-variable" constructor in the sense of Reynolds [82]. We give its syntax and semantics.

In general RML is obviously Turing-complete since the whole of arithmetic is within the language. To preserve some decidability we therefore eliminate full arithmetic by using only a finite subset of the integers as the values for int, and call the resulting language *finitary* RML. We fix this integer set as $\mathbb{N}_{\leq k}$ for some fixed k. In this restricted fragment we let $k + 1 = k$ and $0 - 1 = 0$. For the rest of this thesis when we write to RML we are talking of this finitary fragment.

Types. Types of RML will be made from ground types of int and unit, which represent integers and commands respectively, and the variable type $\mathsf{int\ ref}$. As the int and unit types will be very similar in their behaviour for our purposes, will often use β to range over int and unit.

$$\frac{}{\Gamma \vdash () : \mathsf{unit}} \qquad \frac{i \in \mathbb{N}}{\Gamma \vdash i : \mathsf{int}} \qquad \frac{}{\Gamma, x : \theta \vdash x : \theta}$$

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

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

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

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

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

$$\frac{\Gamma \vdash M : \mathsf{int} \qquad \Gamma \vdash N : \mathsf{unit}}{\Gamma \vdash \mathbf{while}\, M \,\mathbf{do}\, N : \mathsf{unit}}$$

Figure 6.1: Syntax of finitary RML

Types are then constructed from these in the normal way, using the $\to$ operator. That is, they are generated by the grammar:

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

The *order* of a type, which measures the nested-ness of $\to$ in the type, is given by: $\mathbf{ord}(\mathsf{int}) = \mathbf{ord}(\mathsf{unit}) = 0$, $\mathbf{ord}(\mathsf{int\ ref}) = 1$, and $\mathbf{ord}(\theta \to \theta') = max(\mathbf{ord}(\theta) + 1, \mathbf{ord}(\theta'))$. The *arity*, which measures how many arguments the type has, is given by $\mathbf{arity}(\mathsf{int}) = \mathbf{arity}(\mathsf{unit}) = 0$, $\mathbf{arity}(\mathsf{int\ ref}) = 1$, and $\mathbf{arity}(\theta \to \theta') = \mathbf{arity}(\theta') + 1$.

Syntax. The syntax and typing rules of RML terms is given inductively by the rules in Fig. 6.1. Note that although we only include the **succ**() and **pred**() arithmetic operations, addition and multiplication are easily definable. We will write $\mathbf{let}\, x = M \,\mathbf{in}\, N$ as syntactic sugar for $(\lambda x.N)M$, and write $M; N$ for $\mathbf{let}\, x = M \,\mathbf{in}\, N$ where x is chosen to be fresh in N.

Semantics. We present the standard operational semantics, which is a "big-step" relation that uses *stores* [62] to capture the behaviour of variables. Let L range over a countable set of *locations*, then a store is just a finite partial function $s : L \rightharpoonup \mathbb{N}_{\leq k}$. For $l \in L$ and $i \in \mathbb{N}_{\leq k}$ 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 values in L on which s is defined. Having fixed a set of locations, the operational semantics is defined inductively on pairs (s, M), where s is a store and M a closed term, by the rules presented in Fig. 6.2. These reductions reduce terms to values, 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 values inside, $\mathbf{mkvar}(V_1, V_2)$.

6.2 Game Semantics

An operational semantics defines a programming language by describing how terms are evaluated by the interpreter of the language. This can take the form either of a "big-step" or "natural" semantics [46] that give a relation on how terms reduce to canonical forms, as we saw for RML above, or a "small-step" semantics that describe each step of the computation the interpreter of the language would perform [80]. However, whilst such operational semantics are crucial in defining a language, they are not always ideal tools to analyse programs. If we consider the RML semantics above, it was seen that any term of the form $\lambda x.M$ is already in canonical form according to the semantics, and so cannot be analysed further without providing a specific input. More generally, the reductions ascribed by the semantics are dependent on the current interpreter state: in the case of the RML semantics, this is the current store.

A denotational semantics, by contrast, assigns to terms of a programming language a mathematical object that represents the term's meaning, independent of any particular context or interpreter state. Denotational semantics were initially developed by Strachey and Scott as a tool for program analysis [86, 87], with the denotation of a term being a function mapping input to output. Crucially, denotational semantics are generally compositional, and so the denotation of a term is constructible from the denotation of its subterms. A good denotational semantics can then be shown to faithfully represent operational semantics of the language, having properties such

$$V ::= \ () \mid i \mid l \mid \lambda x^{\theta}.M \mid \mathbf{mkvar}(V_1, V_2)$$

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

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

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

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

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

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

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

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

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

Figure 6.2: Semantics of RML

$\mathbb{N} \Rightarrow \mathbb{N}$

Environment: Output?

Program: Input?

Environment: 4

Program: 7

Figure 6.3: Example play of $\lambda x.x + 3$

as soundness – that terms that can give different outputs on the same input have distinct denotations – and full abstraction – that terms have distinct denotations if and only if they can give different outputs on the same input.

Game semantics are a style of denotational semantics that represents computation as interaction between the program and its environment. To give an example, we consider the program $\lambda x.x + 3$ in an environment that requests output with input 4. The interaction begins with the environment requesting output, to which the program requests input. Once the environment provides the input (4, in this case), the program can answer the initial request for output with 7. We give a pictorial representation of this interaction in Fig. 6.3.

We call these sorts of interaction "plays". Whilst in game semantics computation is represented by plays, the actual denotation of a term is a strategy for how the program should interact with the environment, given any possible set of inputs. In the case of the term $\lambda x.x + 3$, this will consist of plays following the same structure as that above, with the program always returning 3 more than the input.

The rest of this section describes how the game semantic denotation of RML terms is constructed.

6.2.1 Games, plays, and strategies

Game semantics for the call-by-name setting [43, 5] differ somewhat from the call-by-value setting [6, 38], and the example we presented in Fig. 6.3 is representative of the call-by-name style of games. As for the rest of this thesis we are concerned with the call-by-value language RML, we just give game semantics in the call-by-value style.

Abramsky and McCusker gave call-by-value game semantics using a categorical construction to translate from the call-by-name setting to call-by-value, in the style of Moggi [60]. Although different in approach, the resulting games are very similar to the direct construction for call-by-value games given by Honda and Yoshida in [38]. As the games by Honda and Yoshida are more concrete, they lend themselves more readily to the kind of algorithmic constructions we intend to perform, and so we present games in this style.

Definition 6.2.1. A *prearena* is a triple $(M_A, \vdash_A, \lambda_A)$ where M_A is the set of moves, $\vdash_A \subseteq (M_A \uplus \{*\}) \times M_A$ is the enabling relation, and $\lambda_A : M_A \to \{O, P\} \times \{Q, A\}$ is a labelling function indicating whether each move belongs to the environment/opponent (O), or program/player (P), and whether it is a question (Q) or answer (A) move. We will write λ_A^{OP} for the restriction of λ_A to the $\{O, P\}$-component, and similarly λ_A^{QA}.

The enabling relation must satisfy the properties that the only moves enabled by $*$ are O-questions, that all moves enabled by O-moves are P-moves and vice-versa, and that all A-moves are enabled by Q-moves.

Prearenas define the games that will be played in representing computation. For RML's semantics we do not distinguish between prearenas and the games they define, and so will use the terms prearena and games interchangeably. The enabling relation describes which moves are valid responses to which other moves, whilst those moves enabled by $*$ are the initial moves.

We will often represent prearenas graphically. An example is shown in Fig. 6.4. The moves at the top of the diagram are the initial moves, and the edges between moves denote the enabling relation. The move occurring lower down is enabled by the higher move. The ownership of a move is completely determined by the fact that O-moves can only enable P-moves and vice-versa. Sometimes whether a move is a question or answer is determined by the structure of the arena, but we will often denote moves by letters q or a for questions and answers respectively.

Definition 6.2.2. Given a prearena $A = (M_A, \vdash_A, \lambda_A)$, a *justified sequence* on A is a sequence of moves from M_A such that each non-initial move m in the sequence has a *justification pointer* from that move to a prior move n in the sequence such that

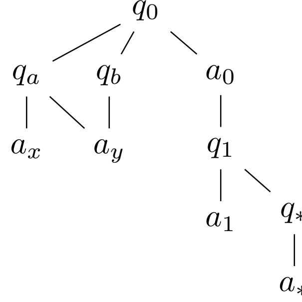
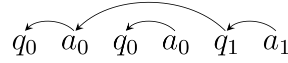


Figure 6.4: Example prearena

$n \vdash_A m$. In this case we say that n *justifies* m, and if we can reach n' by following a sequence of pointers from m we say that n' *hereditarily justifies* m.

Example 6.2.1. We consider a simple prearena with enabling relation $q_0 \vdash a_0 \vdash q_1 \vdash a_1$ and no other moves. (Note that the rules on the enabling relation in this case determine λ for this prearena.) A simple example justified sequence in this prearena is shown below:



Note that the pointer shows which instance of a_0 justifies the q_1-move.

Definition 6.2.3. A *play* is a justified sequence of moves such that the following conditions also hold:

- **Alternation.** O and P alternate moves. That is, if n and m are adjacent moves in the sequence then $\lambda^{OQ}(n) \neq \lambda^{OQ}(m)$.

- **Well-bracketing.** We say that an answer move a answers the question move that justifies it. Well-bracketing requires that whenever an answer is played, it is answering the most recent unanswered question.

- **Visibility.** We define the view function by $\mathsf{view}(\epsilon) = \epsilon$, $\mathsf{view}(m) = m$ where m is initial, and $\mathsf{view}(s \ m \ s' \ n) = \mathsf{view}(s) \ m \ n$. Visibility requires that if $s \ m$ is a prefix of the sequence then the justifier of m occurs in $\mathsf{view}(s)$.

- **Well-opening.** At most one initial move is made (in which case it must be the first move of the sequence).

With plays modelling computation, it is natural to ask when a computation has finished, or what a non-terminating computation looks like. We therefore say a play is *complete* if all questions in the play have been answered, and can think of the complete plays as representing those where the computation could end, as all of O's questions have been answered. However, note that since answers can enable questions, a complete play is not necessarily maximal.

Although computation will be modelled by plays on prearenas, we will model types by the closely related notion of arenas. This definition will allow us to more easily construct the prearenas for specific terms.

Definition 6.2.4. An *arena* is defined exactly as a prearena, except that the initial moves must be P-answers rather than O-questions.

Finally, we get to the definition of strategies. These will be the denotation of a program term. Intuitively, strategies tell P how to play, given the play so far. They can be thought of as functions from odd-length prefixes to the next move. We will define them as sets of even-length plays, the idea being that P will always choose their next move so that the resulting play is in the strategy.

Definition 6.2.5. A *strategy* σ for a prearena A is a non-empty set of even-length plays of A satisfying the following conditions:

- **Even-prefix closure.** If $s\,m\,n \in \sigma$ then $s \in \sigma$; and
- **Determinism.** If $s\,m \in \sigma$ and $s\,n \in \sigma$ then $m = n$.

As we will often be less interested in the behaviour of strategies that are non-terminating, we will sometimes want to consider only the complete plays of a strategy. Given a strategy σ we write $\mathbf{comp}(\sigma)$ for the complete plays in σ.

6.2.2 Game semantics for RML

Having defined the necessary game semantic notions, we now describe how the denotation of an RML term is constructed.

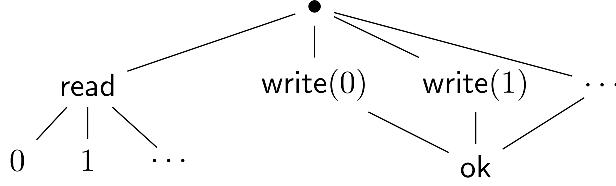


Figure 6.5: Arena for int ref

6.2.2.1 Games for RML sequents

As mentioned previously, the denotation of an RML term will be a strategy. The first step to defining the strategy is to define the game it is a strategy for. Given a term $\Gamma \vdash M : \theta$, the prearena will be determined by θ and the types in Γ. To describe how the prearena is constructed we first make use of some arenas that describe the types individually, and then some useful constructions on these arenas. This will allow us to define a function $[\![-]\!]$ that takes a type and returns its representative arena.

Arenas for ground types. The type unit will be represented by the simple arena, $[\![\mathsf{unit}]\!]$, consisting of just an initial move $\bullet$ (which is a P-answer), and no other moves. $[\![\mathsf{int}]\!]$ is a similar arena, consisting of an initial move i for each $i \in \mathbb{N}_{\leq k}$ and no other moves.

The arena for int ref is slightly more complex, reflecting the fact that it is reminiscent of a first-order function, taking a read- or write-request, and returning an answer. The arena $[\![\mathsf{int\ ref}]\!]$ therefore contains an initial move $\bullet$ which is a P-answer, which enables O-questions read and write(i) for each $i \in \mathbb{N}_{\leq k}$. The read move enables a P-answer i for each $i \in \mathbb{N}_{\leq k}$, whilst the write($i$) moves all enable a single move ok. This is shown pictographically in Fig. 6.5.

Constructions on arenas. We complete the translation between types and arenas by defining a function $\Rightarrow$ such that $[\![\theta_1 \to \theta_2]\!] = [\![\theta_1]\!] \Rightarrow [\![\theta_2]\!]$. Given arenas A and B with initial moves I_A and I_B respectively, $A \Rightarrow B$ is constructed by taking a new initial PA-move $\bullet$, which justifies each move in I_A, each of which is converted (from a PA-move) to an OQ-move. The other moves in A have their λ^{OP}-value switched. The moves in I_B are now each justified by each move in I_A.

We will also define a useful construction on arenas, $\otimes$, that gives a sort of product. Intuitively, $A \otimes B$ is the union of A and B with initial moves combined pairwise, allowing a play to be an interleaving of a play from A and a play from B. Formally,

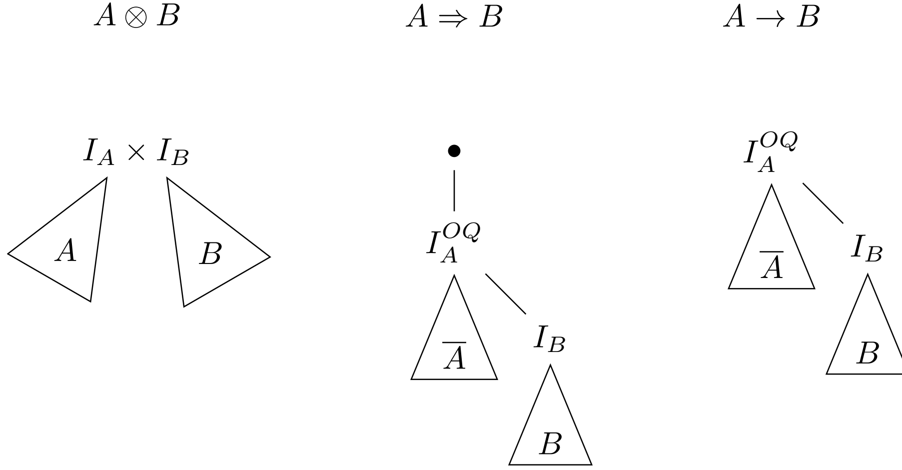


Figure 6.6: Arena and prearena constructions

for arenas A and B with initial moves I_A and I_B respectively, $A \otimes B$ is defined to have initial moves $I_A \times I_B$. The rest of A and B are unchanged except that a move $m \in M_A \backslash I_A$ justified by $i_A \in I_A$ is now justified by all moves in $\{i_A\} \times I_B$, and similarly for moves in $M_B \backslash I_B$.

Prearenas for RML **sequents.** Finally, we give the method for constructing a prearena from an RML sequent $x_1 : \theta_1, \ldots, x_n : \theta_n \vdash M : \theta$. In particular, we give a construction $\to$ taking two arenas and producing a prearena so that $[\![x_1 : \theta_1, \ldots, x_n : \theta_n \vdash M : \theta]\!]$ is given by $([\![\theta_1]\!] \otimes \cdots \otimes [\![\theta_n]\!]) \to [\![\theta]\!]$.

$A \to B$ is essentially the same as $A \Rightarrow B$, but without the initial PA-move. This means the initial moves of $A \to B$ are the initial moves of A, which are now O-questions, and which justify the initial moves of B. B is otherwise unchanged, whilst O-moves in A become P moves and vice-versa.

Given games A and B, the constructions $A \otimes B$, $A \Rightarrow B$, and $A \to B$ are shown in Fig. 6.6. Where the λ-value of moves are altered the new value has been marked in superscript, and we have written $\overline{A}$ for the game A with the ownership of the moves reversed.

6.2.2.2 Strategies for RML terms

We have now described how, given an RML-term $\Gamma \vdash M : \theta$, the prearena corresponding to the type of the term, $[\![\Gamma \vdash \theta]\!]$, is constructed. We now turn our attention to the strategy representing M, which we will denote $[\![\Gamma \vdash M]\!]$ (or simply $[\![M]\!]$ when Γ

is clear).

The strategy for a free variable is known as the copycat strategy. P will simply reply to O's move by playing the same move in a different component. More precisely, if the term is $\Gamma, x : \theta \vdash x : \theta$, the prearena will be of the form $[\![\Gamma]\!] \otimes [\![\theta_1]\!] \to [\![\theta_2]\!]$ where the subscripts are used to differentiate between the two copies. Note that $[\![\theta_1]\!]$ and $[\![\theta_2]\!]$ have had the ownership of the moves inverted relative to one another. O's initial move will have a component corresponding to an initial move of $[\![\theta_1]\!]$, which P can copy into $[\![\theta_2]\!]$. O can then only play a move in $[\![\theta_2]\!]$, as all other enabled moves are P-moves, and P can copy this move into $[\![\theta_1]\!]$, justified by the initial move. The strategy continues with P copying O's move into whichever of $[\![\theta_1]\!]$ or $[\![\theta_2]\!]$ O didn't just play in.

The strategies for the constants of RML can be represented by regular expressions, and these are shown in Fig. 6.7. We use subscripts to indicate which part of the prearena each move comes from, and we omit the justification pointers as these can be reconstructed from the underlying moves. In fact, these regular expressions do not precisely describe the strategies, though they can be said to generate them: the strategies will of course be even-prefix-closed, and so even prefixes of strings matching the regular expression will be in the strategy. Further, the prearenas generally permit O to open additional threads by repeating non-initial questions. Therefore the strategies, with the exception of $[\![\mathbf{ref}\,]\!]$ which is fully described in the figure, will actually permit any play where the view at each point matches an even-prefix of the regular expression.

$[\![\Gamma \vdash \mathbf{if}\, M \,\mathbf{then}\, N_1 \,\mathbf{else}\, N_2 : \mathsf{unit}]\!]$ is a fairly straightforward concatenation of $[\![M]\!]$ together with $[\![N_1]\!]$ or $[\![N_2]\!]$ depending on whether $[\![M]\!]$ returns 0 or not. The only subtlety is that the final move of $[\![M]\!]$, when P would return 0 or another number, is omitted, and the first of P's moves in $[\![N_1]\!]$ or $[\![N_2]\!]$ is played immediately instead.

Similarly, $[\![\Gamma \vdash \mathbf{while}\, M \,\mathbf{do}\, N]\!]$ runs as $[\![M]\!]$ until a value would be returned, and if that value is not 0 it runs as $[\![N]\!]$ until the final move would be played. This repeats until 0 would be returned, at which point it plays $\bullet$ answering O's initial question.

We now just have the application and lambda abstraction strategies to describe. We begin with the λ-abstraction case. Suppose we have $\Gamma, x : \theta' \vdash M : \theta$, then $[\![M]\!]$ is played on $[\![\Gamma]\!] \otimes [\![\theta']\!] \to [\![\theta]\!]$, so the initial moves will contain a component correspond-

$$
\begin{aligned}
&[\![\Gamma \vdash () : \mathsf{unit}]\!] \;=\; i_\Gamma \bullet \\
&[\![\Gamma \vdash j : \mathsf{int}]\!] \;=\; i_\Gamma\; j \\
&[\![\Gamma \vdash \mathbf{succ}(x) : \mathsf{int}_1]\!] \;=\; \sum_{j \in \mathbb{N}_{\leq k}} j_\Gamma\; (j+1)_1 \\
&[\![\Gamma \vdash \mathbf{pred}(x) : \mathsf{int}_1]\!] \;=\; \sum_{j \in \mathbb{N}_{\leq k}} j_\Gamma\; (j-1)_1 \\
&[\![\Gamma \vdash \mathbf{ref} - : \mathsf{int}_1 \rightarrow_2 \mathsf{int\ ref}_3]\!] \;=\; i_\Gamma\; \bullet_2\; j_1\; \bullet_3\; (\mathsf{read}_3\; j_3)^* \Big(\sum_{n \in \mathbb{N}_{\leq k}} \mathsf{write}(n)_3\; \mathsf{ok}_3\; (\mathsf{read}_3\; n_3)^* \Big)^* \\
&[\![\Gamma \vdash !- : \mathsf{int\ ref}_1 \rightarrow_2 \mathsf{int}_3]\!] \;=\; i_\Gamma\; \bullet_2\; \bullet_1\; \mathsf{read}_1 \sum_{j \in \mathbb{N}_{\leq k}} j_3\; j_1 \\
&[\![\Gamma \vdash - := - : \mathsf{int\ ref}_1 \rightarrow_2 \mathsf{int}_3 \rightarrow_4 \mathsf{unit}_5]\!] \;=\; i_\Gamma\; \bullet_2\; \bullet_1\; \bullet_4 \sum_{j \in \mathbb{N}_{\leq k}} j_3\, \mathsf{write}(j)_1\; \mathsf{ok}_1\; \bullet_5 \\
&[\![\Gamma \vdash \mathbf{mkvar}(-,-) : (\mathsf{unit}_1 \rightarrow_2 \mathsf{int}_3) \rightarrow_4 (\mathsf{int}_5 \rightarrow_6 \mathsf{unit}_7) \rightarrow_8 \mathsf{int\ ref}_9]\!] \\
&\qquad = \; i_\Gamma\; \bullet_4\; \bullet_2\; \bullet_8\; \bullet_6\; \bullet_9 \Big(\mathsf{read}_9\; \bullet_1 \sum_{j \in \mathbb{N}_{\leq k}} j_3\; j_9 \;+\; \sum_{j \in \mathbb{N}_{\leq k}} \mathsf{write}(j)_9\; j_5\; \bullet_7\; \mathsf{ok}_9 \Big)^*
\end{aligned}
$$

Figure 6.7: Strategies of RML constants

ing to an initial move of x's copy of $[\![\theta']\!]$, and we will write these moves as (i_Γ, i_x). Now $[\![\Gamma \vdash \lambda x.M : \theta' \rightarrow \theta]\!]$ is played on $[\![\Gamma]\!] \rightarrow [\![\theta']\!] \Rightarrow [\![\theta]\!]$, so initial moves will just be the i_Γ-component of initial moves of $[\![M]\!]$. The strategy $[\![\lambda x.M]\!]$ will, after O's initial move i_Γ, request an initial x-move. Once such a move, i_x is provided, P plays as $[\![M]\!]$ when provided with initial move (i_Γ, i_x). However, since i_x is no longer initial, O will be able to play it (or other initial moves of $[\![\theta']\!]$) multiple times. Each time O does it opens a new "thread" which P plays in as $[\![M]\!]$. The strategy then consists of interleavings of these threads. Only O has the ability to switch between these threads, though he/she can only do so in ways that respect the visibility condition.

Finally, we look at application. Suppose we have $\Gamma \vdash M : \theta_1 \rightarrow \theta_2$ and $\Gamma \vdash N : \theta_1$. The strategy $[\![\Gamma \vdash M\, N : \theta_2]\!]$ begins as $[\![M]\!]$, until the (unique) occurence of the PA-move $\bullet$ which is the initial move for $[\![\theta_1 \rightarrow \theta_2]\!]$ would be played. At this point P plays as $[\![N]\!]$, until the initial θ_1-move would be played. P takes this move as the reply to $\bullet$, and continues play as per $[\![M]\!]$. In general whenever a θ_1-move, m, would be played in $[\![M]\!]$, control passes to $[\![N]\!]$ until it would make a reply to m. Then control passes back to $[\![M]\!]$, which plays as if it played m and received $[\![N]\!]$'s reply.

This completes the definition of the game semantic denotation of RML terms. We next look at what these semantics let us do.

6.3 Observational equivalence

Observational equivalence (OE) – also known as contextual equivalence – is the problem of whether two program-fragments are completely interchangeable without changing any externally observable results. We give a formal definition in Definition 6.3.1. OE is a natural notion of program equivalence, a key problem in verification [34], and Part II of this thesis is principally concerned with which fragments of RML have a decidable OE problem.

Definition 6.3.1. For an RML term M we write $M \Downarrow$ if there exists s, 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* just if for all contexts $C[-]$ such that $\Gamma \vdash C[M], C[N] : \mathsf{unit}$, $C[M] \Downarrow$ iff $C[N] \Downarrow$.

6.3.1 Deciding observational equivalence of RML

As discussed in Section 1.2, observational equivalence for Idealized Algol [82] is decidable for terms of order up to 3, and undecidable for terms of order 4 and higher. Observational equivalence of RML is a less well-studied area, and the fragments for which observational equivalence is decidable are less well understood. Unlike in the IA case, the type-theoretic order does not give a clean divide between decidable and undecidable fragments: there are decidable fragments with order 3 free variables, and undecidable fragments using only types of order at most 2. These results and those we present in this thesis, like for Idealized Algol, rest on the full abstraction result linking observational equivalence to equivalence of the game semantic denotation of terms. This result is given in Theorem 6.3.1.

Theorem 6.3.1 (Abramsky and McCusker [5, 6])**.** *If* $\Gamma \vdash M : \theta$ *and* $\Gamma \vdash N : \theta$ *are* RML *terms then* $\Gamma \vdash M \cong N$ *iff* $\mathbf{comp}(\llbracket M \rrbracket) = \mathbf{comp}(\llbracket N \rrbracket)$.

Prior to the work in this thesis, several decidability and undecidability results had been obtained for observational equivalence of RML, in [62, 41, 40]. We recap these here.

Fragment	Type Sequents	Recursion
Decidable		
O-Strict (ExpTime-Complete)	$((\beta \to \ldots \to \beta) \to \beta) \to \ldots \to \beta \vdash (\beta \to \ldots \to \beta) \to \beta$	**while**
O-Strict + Recursion (DPDA-Hard)	$((\beta \to \ldots \to \beta) \to \beta) \to \ldots \to \beta \vdash (\beta \to \ldots \to \beta) \to \beta$	$\beta \to \beta$
RML_{CMA}	$(\beta \to \beta) \to \ldots \to \beta \vdash \beta \to \beta \to \beta$	**while**
Undecidable		
Third-Order	$\vdash ((\beta \to \beta) \to \beta) \to \beta$ $(((\beta \to \beta) \to \beta) \to \beta) \to \beta \vdash \beta$	Ω
Second-Order	$\vdash (\beta \to \beta) \to \beta \to \beta$ $((\beta \to \beta) \to \beta \to \beta) \to \beta \vdash \beta$	Ω
Recursion	Any	$(\beta \to \beta) \to \beta$

Table 6.1: Decidability results for RML prior to work in this thesis

In [62] Murawski used regular expressions, building on work in [31, 32], to show decidability of a fragment of RML consisting of types of the form $(\beta \to \beta) \to \beta \vdash (\beta \to \beta) \to \beta \to \beta$, though this required that all subterms of the term were also in this fragment. Murawski also showed that observational equivalence was undecidable at the second-order type $\vdash (\beta \to \beta) \to (\beta \to \beta) \to \beta$.

Work in [41] looked at a fragment of RML, $\text{RML}_{\text{O-Str}}$, containing second-order terms, in which all O-pointers were determined by the underlying sequence of moves. By reduction to visibly pushdown automata, they showed that OE was decidable for this fragment. Further work, presented in [40], identified a fragment, RML_{CMA}, which could be reduced to class memory automata. This fragment permitted first-order types with two arguments. These fragments, and their reductions, are discussed in more detail in the rest of this section.

We first present Table 6.1, from [40], summarising the known decidability results for RML prior to work presented in this thesis, and we briefly discuss observational equivalence in languages highly related to RML.

An Aside: Observational equivalence of other call-by-value languages. Observational equivalence has been studied in other call-by-value languages using game semantic methods. In particular the OE problem for Reduced ML, which may be considered to be as RML but without the bad-variable constructor, was studied

in [67, 68], and decidability of a fragment containing $\beta \to \beta$ terms was obtained using automata-theoretic methods similar to those used in this thesis [68]. OE has also been studied in a call-by-value setting containing general references, with a full abstraction result shown in [69]. A restriction of this to "full ground references" (references for ground types other references only) was studied in [70], in which OE was shown undecidable at types $\vdash \beta \to \beta \to \beta$ and $\vdash ((\beta \to \beta) \to \beta) \to \beta$. Relational methods have also been used to study observational equivalence of similar languages, for instance in [79, 50, 8, 26]. However, this approach has not led to any general decidability results.

6.3.1.1 Canonical forms

In deciding both $\mathrm{RML}_{\text{O-Str}}$ and $\mathrm{RML}_{\mathrm{CMA}}$, to simplify the inductive constructions a kind of normal form for RML was used. We recap this here.

Definition 6.3.2. An RML term is in *canonical form* if it is generated by the following grammar:

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

Theorem 6.3.2 (Hopkins, [40]). *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* $[\![\Gamma \vdash M]\!] = [\![\Gamma \vdash N]\!]$.

Crucially, Theorem 6.3.2 means that we can assume terms to be in canonical form when making these constructions. However, it is worth mentioning that whilst the translation to canonical form is effective, it is non-elementary in complexity.

6.3.1.2 Recap of deciding O-strict RML

We recap in a little more detail how $\mathrm{RML}_{\text{O-Str}}$ was shown to be decidable, as many of these techniques and ideas will be used later in this thesis.

Informally, the O-strict fragment of RML, written $\mathrm{RML}_{\text{O-Str}}$, is defined using the idea of *short types*, a short type being one with order at most 2 and arity at most 1. Hence, short types are of the form $(\beta \to \cdots \to \beta) \to \beta$. $\mathrm{RML}_{\text{O-Str}}$ is then defined

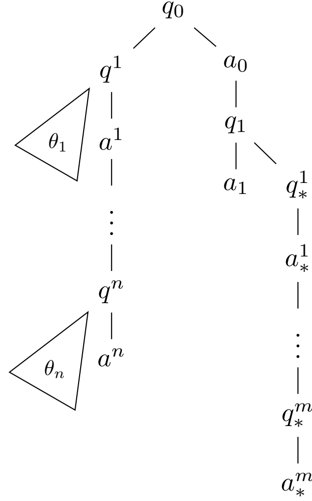


Figure 6.8: Shape of prearena for $\theta_1 \to \cdots \to \theta_n \to \beta \vdash (\beta \to \cdots \to \beta) \to \beta$

to consist of terms of short type such that the arguments of any free variables in the term are also short. In particular, since O's only question-moves are justified by moves directly justified by the initial move, visibility enforces that the O-pointers of the plays are uniquely determined by the underlying sequence of moves – hence the name of the fragment. This will be crucial in that reducing the strategies to languages over an alphabet will not require these O-pointers to be encoded.

Formally, $\mathrm{RML}_{\text{O-Str}}$ is defined to consist of terms of the form:

$$x_1 : \Theta_3, \ldots, x_n : \Theta_3 \vdash M : \Theta_2$$

where

$$\Theta_0 ::= \mathsf{unit} \mid \mathsf{int} \qquad\qquad \Theta_1 ::= \Theta_0 \mid \Theta_0 \to \Theta_1 \mid \mathsf{int\ ref}$$

$$\Theta_2 ::= \Theta_0 \mid \Theta_1 \to \Theta_0 \mid \mathsf{int\ ref} \qquad\qquad \Theta_3 ::= \Theta_0 \mid \Theta_2 \to \Theta_3 \mid \mathsf{int\ ref}$$

The shape of the prearenas for terms in this fragment is shown in Fig. 6.8. We briefly discuss what plays in these prearenas can look like. Play starts with O making the inital q_0-move, at which point P can interrogate $[\![\Gamma]\!]$. At some point P plays the move a_0, answering the initial question. At this point O can play q_1, beginning a new "thread". Once the thread has begun P can again interrogate $[\![\Gamma]\!]$, as well as now being able to play q^1_*, or end this thread by playing a_1. Whenever P plays q^1_* (or another q^i_*-move) O has the option to start a new thread by playing a q_1-move,

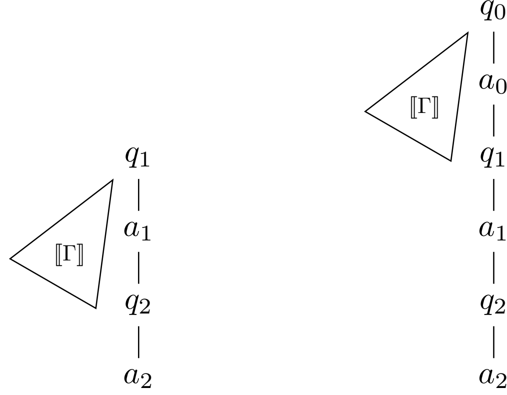
or to simply play a_*^1. If O does start a new thread, that thread must be played to completion before the old thread can be revisited, at which point O again has the same option. Only once O plays a_*^1 (or a_*^i) does play revert to that initial thread, at which point P may play a_1, another q_*^i-move, or a move in $[\![\Gamma]\!]$, continuing the thread. Note that a thread started after a q_*^i-move may also have a q_*^i-move played in it, and this would allow a new thread to be started, giving rise to an unbounded number of "nested" threads.

The key result of $\mathrm{RML}_{\text{O-Str}}$ is recapped here:

Theorem 6.3.3 (Hopkins et al. [41]). *For terms $\Gamma \vdash M$ in $\mathrm{RML}_{\text{O-Str}}$, there is an effective method of constructing a visibly pushdown automaton $\mathcal{A}_{\Gamma\vdash M}$ such that given two terms M and N, $\Gamma \vdash M \cong N$ iff $\mathcal{L}(\mathcal{A}_{\Gamma\vdash M}) = \mathcal{L}(\mathcal{A}_{\Gamma\vdash N})$.*

Since equivalence of VPA is ExpTime-complete, this means observational equivalence for this fragment is decidable. Unsurprisingly, the proof gives a better result than this: it provides a procedure that, given $\Gamma \vdash M$ in $\mathrm{RML}_{\text{O-Str}}$, constructs a VPA recognising a representation of $[\![\Gamma \vdash M]\!]$. This representation uses the underlying moves of the prearena as the alphabet. Above we described how playing a q_*^i-move let O then begin new threads which must be completed before the corresponding a_*^i-move could be played. Hence it is the q_*^i-moves that correspond to pushes in the automaton, and a_*^i-moves that correspond to pops.

However, just giving the underlying string of moves of the plays is insufficient: the pointers of P-questions need to be represented. (Since answer-moves' pointers are determined by well-bracketing, and this is the O-strict fragment, this is sufficient.) These P-pointers are encoded using a "tagging" of moves by marking the source and target of a pointer by respective markings. In more detail, given an arena with moves $\{a_1, \ldots, a_n\}$ we take an alphabet with letters $\{a_1, \ldots, a_n, \overset{\bullet}{a}_1, \ldots, \overset{\bullet}{a}_n, \overset{\circ}{a}_1, \ldots, \overset{\circ}{a}_n\}$. A P-pointer $s\ m\ s\ m'$ is then encoded with the string $s\ \overset{\bullet}{m}\ t\ \overset{\circ}{m}'$. For each play, the language will have several words: each word in the language encodes only one P-pointer, but the language includes a word for each P-pointer to be encoded. Because P's strategy must be deterministic, by beginning at the beginning of a word, and examining the other words in the language whenever a P-pointer ambiguity arises, it is possible to reconstruct all of the pointers in the play that word partially represents. That is, representing one pointer in each word is enough to uniquely reconstruct all



$[\![\Gamma \vdash \beta \to \beta]\!]$ $\qquad$ $[\![\Gamma \vdash \beta \to \beta \to \beta]\!]$

Figure 6.9: Prearenas for $\Gamma \vdash \beta \to \beta$ and $\Gamma \vdash \beta \to \beta \to \beta$

pointers in the plays from the language. Thus, the finite alphabet is able to represent arbitrary P-pointers in a word.

6.3.1.3 RML$_{\text{CMA}}$

We also briefly recap how RML$_{\text{CMA}}$ was decided in [40], as these ideas will be crucial in Chapter 7.

The fragment RML$_{\text{CMA}}$ was defined to contain first-order terms with arity at most 2, with free variables of order at most 2 such that each argument in the free variable's type has arity at most 1. That is, it consisted of types of the form:

$$(\beta \to \beta) \to \cdots \to (\beta \to \beta) \to \beta \vdash \beta \to \beta \to \beta.$$

In previous work [62] it had been shown that terms of type $\beta \to \beta$ could have their strategies faithfully described by regular expressions, and hence had a decidable observational equivalence problem. This used the fact that terms of this type are "bistrict", meaning all pointers are determined by the underlying moves.

The arenas for the bistrict fragment and RML$_{\text{CMA}}$ are shown in Fig. 6.9. Plays in RML$_{\text{CMA}}$ consist of O playing q_0 after which P may interrogate Γ (and this interrogation must be completed) before playing a_0. After this, O plays q_1, starting a thread corresponding to a term of type $\beta \to \beta$. Then, whenever P plays an a_1 or a_2-move,

O has the option to create a new thread (by playing a q_1-move), or switching to an existing q_1-thread (by playing a q_2-move).

In [40] Hopkins and Murawski used deterministic class memory automata to recognise the strategies of terms in $\mathrm{RML}_{\mathrm{CMA}}$. The key idea of this was to use data values to identify each $\overset{\curvearrowleft}{q_1\ a_1}$-thread. As described above, the plays in this fragment are interleavings of the $\overset{\curvearrowleft}{q_1\ a_1}$-threads corresponding to terms of type $\beta \to \beta$, and we saw in Part I how CMA are ideally suited for recognising interleavings of regular languages. In this instance the class memory function is used to remember the current state of each thread (including the values of variables local to that thread, for instance).

In fact, although [40] used full (i.e. non-weak) class memory automata, and showed that equivalence of deterministic CMA is reducible to emptiness of non-deterministic CMA, this is not necessary and better complexity can be obtained. Because threads can only be changed after a_i-moves – when the play is complete – no local properties have to be checked at the end of the run, and so weak class memory automata are sufficient. This reduces the complexity of checking OE (for terms in canonical form) to 2-EXPSPACE[1].

[1]This bound is obtained by an exponential blow up in states in constructing the WCMA, and the fact that WCMA have an EXPSPACE-complete equivalence problem.

Chapter 7

Decidable fragments of $\mathrm{RML}_{2\vdash 1}$

In this chapter, we consider the RML fragment $\mathrm{RML}_{2\vdash 1}$, consisting of all first-order terms with second-order free-variables. Whilst we are unable to prove observational equivalence decidable for the whole fragment, we identify two subfragments for which the terms may be represented by deterministic weak NDCMA, and which hence have a decidable observational equivalence problem.

We begin by considering the whole of $\mathrm{RML}_{2\vdash 1}$, and briefly discuss the shape of plays in this fragment. We then look at the P-Strict subfragment of $\mathrm{RML}_{2\vdash 1}$. This consists of all types within $\mathrm{RML}_{2\vdash 1}$ for which the P-pointers of the plays are determined by the underlying moves. We show a reduction from terms in this subfragment to deterministic weak NDCMA. We then consider $\mathrm{RML}_{2\vdash 1}^{\mathrm{LHS-O-str}}$, which is the subfragment such that the free variables' types are O-Strict. This fragment extends the fragment decidable by weak CMA by including all first-order types on the RHS. Again, we show a reduction to deterministic weak NDCMA for this fragment, thus obtaining decidability of observational equivalence for this fragment.

This chapter is based on material first presented in [18], and is joint work with Andrzej Murawski and Luke Ong.

7.1 $\mathrm{RML}_{2\vdash 1}$

$\mathrm{RML}_{2\vdash 1}$ is defined to consist of all terms-in-context $x_1 : \theta_1, \ldots, x_n : \theta_n \vdash M : \theta$ where θ is a first-order type, and $\theta_1, \ldots, \theta_n$ are all types of order at most 2. The shape of prearenas in this fragment is shown in Fig. 7.1

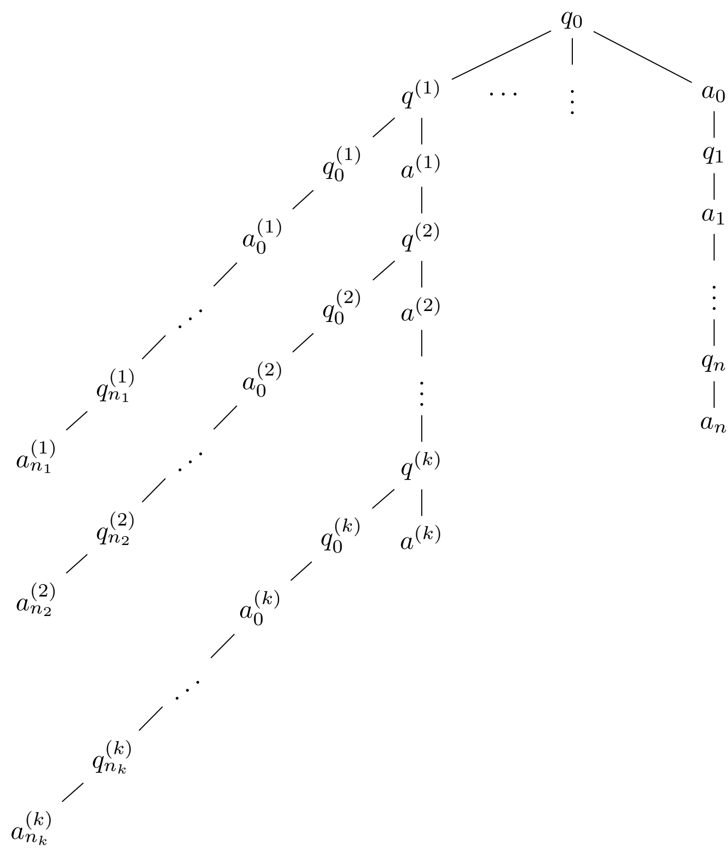


Figure 7.1: Shape of prearenas in RML$_{2\vdash 1}$

Similar to the plays of $\mathrm{RML}_{\mathrm{CMA}}$, plays in this fragment consist, after the initial $\overset{\frown}{q_0 \ a_0}$-moves, of interleavings of $\overset{\frown}{q_1 \ a_1}$-threads. $\overset{\frown}{q_1 \ a_1}$-threads then consist, after the $\overset{\frown}{q_1 \ a_1}$-moves, of interleavings of $\overset{\frown}{q_2 \ a_2}$-threads, and so on. The current thread can only change after an a_i-move, when O has the choice of which thread to play, by playing a suitably justified q_j-move.

Whilst this is relatively straightforward, how these $\overset{\frown}{q_i \ a_i}$-pairs interact with the moves corresponding to the LHS types is rather more complex. Whenever P plays a $q^{(i)}$ move, O has the opportunity to play $q_0^{(i)}$, starting a new thread which may be played in a similar manner to the RHS type. However, O may also go back to previous $q_0^{(j)}$-threads, playing a move $q_{j'}^{(j)}$, or starting a new such thread by playing $q_0^{(j)}$. Such moves, though, may only be made subject to the visibility condition. This means that the hereditarily justifying move $q^{(j)}$ must hereditarily justify the most recent (unanswered) $q^{(i)}$ move.

Example 7.1.1. Consider the following term of type $\vdash \mathsf{unit} \to \mathsf{unit} \to \mathsf{unit} \to \mathsf{unit}$:

$$M \quad = \quad \lambda x^{\mathsf{unit}}.\mathbf{let}\, c = \mathbf{ref}\, 0\, \mathbf{in}\, \lambda y^{\mathsf{unit}}.\ c := 1;\ \lambda z^{\mathsf{unit}}.\mathbf{if}\, !c = 0\, \mathbf{then}\, \Omega\, \mathbf{else}\, c := 0$$

We briefly describe what $[\![M]\!]$ looks like:

Play starts with O asking the initial question q_0, to which P replies with a_0, the only move available. Now O can start a thread by playing q_1 (essentially providing input for x). P replies with a_1. At this point O can either start another q_1-thread, or play q_2 (providing input for y). After this, O can again either start a new q_1-thread, or return to any existing thread by playing a q_2 or q_3-move. Since there is nothing before the $\lambda x.$ of M, there are no restrictions on how the q_1-threads are interleaved, but we now look at what these q_1-threads look like. In particular, note that at least one $\overset{\frown}{q_2 \ a_2}$-pair must occur between $\overset{\frown}{q_3 \ a_3}$-moves. This is because after a q_3-move P evaluates the term $\mathbf{if}\, !c = 0\, \mathbf{then}\, \Omega\, \mathbf{else}\, c := 0$. Evaluating this twice in a row must cause $!c$ to be 0, leading to divergence. Each q_2-move, though, will cause P to set the value of c to 1, letting the next occurrence of q_3 not lead to divergence.

Hence, $[\![M]\!]$ consists of plays which begin with $\overset{\frown}{q_0 \ a_0}$, and are then followed by an interleaving of q_1-threads, where each q_1-thread has its moved described by the following regular expression: $q_1 a_1\, ((q_2 a_2)^+\, q_3 a_3\,)^*\, (q_2 a_2)^*$. The interleaving can switch between threads after any a_i-move.

As mentioned in Section 6.3.1.3, observational equivalence of terms of type $\beta \to \beta \to \beta$ has been shown decidable by reduction to language equivalence of class memory automata, using data values as identifiers for each $\overset{\curvearrowleft}{q_1 \ a_1}$-thread. The difficulty in extending this approach all types $\beta \to \cdots \to \beta$ is that as the layers grow, the threads taking on a nested structure, and normal data values and CMA are not able to represent this. Nested data CMA, however, are ideally suited to represent this parent-child relationship between data values, and so in this chapter we look at using deterministic weak NDCMA to recognise strategies of RML-terms of these types.

A note on NDCMA in this chapter. In Chapter 4 we aimed to present as clean a definition of nested data CMA as possible. To aid the clarity of inductive constructions of NDCMA in this chapter, we make some slight cosmetic alterations to the NDCMA definition.

In particular we used a dataset consisting of an infinite forest of data values, we instead will use an infinite tree, by linking together all level-1 data values under a single level-0 root data value. (Note that this doesn't change power as the CMF of the level-0 data value can be stored in the present state using a product construction.) Further, in Chapter 4 when a NDCMA transitioned to a state q reading a data value d, the CMF updated to have $f(d) = q$ and $f(d') = q$ where d' is an ancestor of d. For this chapter we will permit the transition function to specify the new value for $f(d)$, and for each $f(d')$. (Like for CMA with labels, a simple product construction would allow this to be encoded into the NDCMA of Chapter 4.)

Hence the transition function for a level-k NDCMA will be of the form $\bigcup_{0 \leq i \leq k} \delta_i$ where each $\delta_i : Q \times \Sigma \times (Q_\perp)^{i+1} \to \mathcal{P}(Q \times Q^{i+1})$. If $(p, \bar{p}) \in \delta(q, a, \bar{s})$ we may write this as $q \xrightarrow{a,\bar{s}} p, \bar{p}$. If the NDCMA is in configuration (q, f) and reads (a, d) it can follow a transition $q \xrightarrow{a,\bar{s}} p, \bar{p}$ if $\bar{s} = (f(pred^i(d)), \ldots, f(pred(d)), f(d))$, and it ends in configuration (p, f') where $(f'(pred^i(d), \ldots, f'(pred(d)), f'(d)) = \bar{p}$ and otherwise f' agrees with f.

7.2 $\mathrm{RML}^{\mathrm{P-str}}_{2\vdash 1}$

Whilst NDCMA are ideal for representing the nesting of threads corresponding to the moves on the RHS of types in $\mathrm{RML}_{2\vdash 1}$, we find that types on the left are somewhat harder to deal with and so we look at restricting the types we permit. In this section

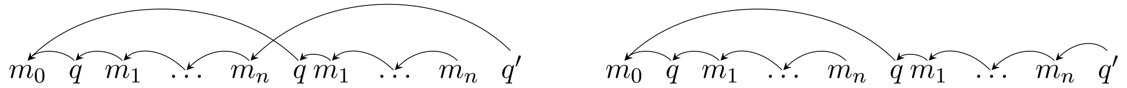
we look at restricting the types to those where we will not need to represent P-pointers (as the P-pointers will be determined by the underlying sequence of moves).

If we consider a type sequent $\Gamma \vdash \theta$ in $\mathrm{RML}_{2\vdash 1}$, we find the P-moves in $[\![\Gamma \vdash \theta]\!]$ corresponding to the type θ (which is order 1) are all answers, and so the pointers for these moves are determined by the well-bracketing condition. The types on the other side of the turnstile, however, can break P-strictness. Suppose there is a type θ' of arity > 1 in Γ. Then the corresponding prearena will contain an enabling sequence $q_0 \vdash q \vdash a \vdash q'$ (where q_0 is an initial move). This means the sequences $q_0\ q\ a\ q\ a\ q'$ and $q_0\ q\ a\ q\ a\ q'$ are both valid plays, differing only in the location of a P-pointer. Hence for $[\![\Gamma \vdash \theta]\!]$ to be P-strict, all types in Γ must be of arity at most 1. In fact, this is a sufficient restriction on $\mathrm{RML}_{2\vdash 1}$ to obtain a P-strict fragment: the resulting prearenas are shown in Figure 7.2, and it can be seen that the only P-questions are justified by the initial move, and hence their pointers will always be determined.

In working out which type sequents for RML lead to prearenas which are P-strict, it is natural to ask for a general characterisation of such prearenas. The following lemma provides exactly that:

Lemma 7.2.1. *A prearena is P-strict iff there is no enabling sequence $q \vdash \cdots \vdash q'$ in which both q and q' are P-questions.*

Proof. Suppose there is such a sequence. Since q is a P-move it is non-initial, so is justified by another move m_0. Suppose the enabling sequence is thus $m_0 \vdash q \vdash m_1 \vdash \cdots \vdash m_n \vdash q'$. Then the following sequences witness the fact that the prearena is not P-strict:



Conversely, suppose a sequence is not P-strict. Then there are two plays, p_1 and p_2 which differ only in the location of a P-pointer. We can assume these plays end with the move with the differing P-pointers, so they are of the form:

$$\bar{s}\ m\ \bar{t_1}\ m\ \bar{t_2}\ m' \quad \text{and} \quad \bar{s}\ m\ \bar{t_1}\ m\ \bar{t_2}\ m'$$

If m' were an answer move well-bracketing would determine the pointer, so it must be a P-question. Now consider the last move, m_0, that hereditarily justifies both copies of m. If m_0 is a P-question, we are done. If it is an O-answer, any moves justified by it must be P-questions, so we are also done. If it is an O-question, the next move in the enabling chain $m_0 \vdash m_1 \vdash \cdots \vdash m_n = m \vdash m'$ must be either a P-question, in which case we are done, or a P-answer, in which case we are also done, as no question move can be answered twice and m_0 is the last move to hereditarily justify both copies of m. Finally, if m_0 is a P-answer, either there is a P-question in the chain $m_1 \vdash \cdots \vdash m$, in which case we are done, or m_1, m_3, $\ldots$ m_n are all O-questions, while m_2, m_4, $\ldots$, m_{n-1} are P-answers. But in this case by visibility requirements both m moves cannot be in the view at the same time.

$\square$

Which type sequents lead to a P-question hereditarily justifying another P-question? It is clear, from the construction of the prearena from the type sequent, that if a free variable in the sequent has arity > 1 or order > 2, the resulting prearena will have a such an enabling sequence, so not be P-strict. Conversely, if a free variable is of a type of order at most 2 and arity at most 1, it will not break P-strictness. On the RHS of the type sequent, things are a little more complex: there will be a "first" P-question whenever the type has an argument of order ≥ 1. To prevent this P-question hereditarily justifying another P-question, the argument must be of arity 1 and order ≤ 2. Hence the P-strict fragment consists of type sequents of the following form:

$$(\beta \to \cdots \to \beta) \to \beta \vdash ((\beta \to \cdots \to \beta) \to \beta) \to \cdots \to ((\beta \to \cdots \to \beta) \to \beta) \to \beta$$

(where $\beta \in \{\mathsf{unit}, \mathsf{int}\}$.)

From results shown in [40, 18], we know that observational equivalence of all type sequents with an order 3 type or order 2 type with order 1 non-final argument on the RHS are undecidable. Hence the only P-strict types for which observational equivalence may be decidable are of the form:

$$(\beta \to \cdots \to \beta) \to \beta \vdash \beta \to \cdots \to \beta$$

or

$$(\beta \to \cdots \to \beta) \to \beta \vdash \beta \to \cdots \to \beta \to (\beta \to \cdots \to \beta) \to \beta$$

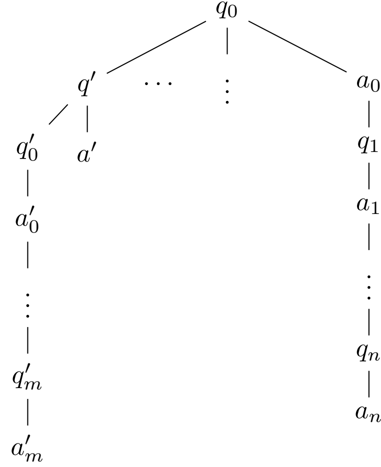


Figure 7.2: Shape of prearenas for $\text{RML}_{2\vdash 1}^{\text{P-Str}}$

We now show that the first of these does lead to decidability. The second of these cases is examined in more detail in Chapters 8 and 9.

7.2.1 Fragment Definition

Definition 7.2.1. The P-Strict fragment of $\text{RML}_{2\vdash 1}$, which we denote $\text{RML}_{2\vdash 1}^{\text{P-Str}}$, consists of typed terms of the form $x_1 : \widehat{\Theta_1}, \ldots, x_n : \widehat{\Theta_1} \vdash M : \Theta_1$ where the type classes Θ_i are as described below:

$$\Theta_0 ::= \mathsf{unit} \mid \mathsf{int} \qquad \Theta_1 ::= \Theta_0 \mid \Theta_0 \to \Theta_1 \mid \mathsf{int\ ref} \qquad \widehat{\Theta_1} ::= \Theta_0 \mid \Theta_1 \to \Theta_0 \mid \mathsf{int\ ref}$$

This means we allow types of the form $(\beta \to \cdots \to \beta) \to \beta \vdash \beta \to \cdots \to \beta$ where $\beta \in \{\mathsf{unit}, \mathsf{int}\}$. Fig. 7.2 shows the shape of prearenas for this fragment. The moves on the right hand side of the prearena diagram correspond to the type on the right of the sequent (i.e. the type of the term), and moves on the left hand side of the diagram correspond to the types on the left of the sequent (i.e. the types of the free variables).

7.2.2 Deciding Observational Equivalence of $\text{RML}_{2\vdash 1}^{\text{P-Str}}$

Our aim is to decide observational equivalence by constructing, from a term M, an automaton that recognises a language representing $\mathbf{comp}(\llbracket M \rrbracket)$. As $\mathbf{comp}(\llbracket M \rrbracket)$ is a set of plays, the language representing $\mathbf{comp}(\llbracket M \rrbracket)$ must encode both the moves and the pointers in the play. Since answer moves' pointers are always determined

by well-bracketing, we only represent the pointers of question moves, and we do this with the nested data values. The idea is simple: if a complete play s is in $[\![M]\!]$ the language $L(\mathbf{comp}([\![M]\!]))$ will contain a word, w, such that the string projection of w is the underlying sequence of moves of s, and such that:

- The initial move takes the (unique) level-0 data value; and
- Answer moves take the same data value as that of the question they are answering; and
- Other question moves take a fresh data value whose predecessor is the data value taken by the justifying move.

For convenience, in the following we may refer to $L(\mathbf{comp}([\![M]\!]))$ as just $L([\![M]\!])$ or simply $[\![M]\!]$.

Of course, the languages recognised by nested data automata are closed under automorphisms of the data set, so in fact each play s will be represented by an infinite set of data words, all equivalent to one another by automorphism of the data set.

Theorem 7.2.2. *For every typed term $\Gamma \vdash M : \theta$ in $\mathrm{RML}^{\text{P-Str}}_{2\vdash 1}$ that is in canonical form we can effectively construct a deterministic weak nested data class memory automaton, $\mathcal{A}^M$, recognising $L([\![\Gamma \vdash M]\!])$.*

Proof. We prove this by induction over the canonical forms. We note that for each canonical form construction, if the construction is in $\mathrm{RML}^{\text{P-Str}}_{2\vdash 1}$ then each constituent canonical form must also be. For convenience of the inductive constructions, we in fact construct automata $\mathcal{A}^M_\gamma$ recognising $[\![\Gamma \vdash M]\!]$ restricted to the initial move γ.

The shape of the pre-arena for terms $[\![\Gamma \vdash M]\!]$ in $\mathrm{RML}^{\text{P-Str}}_{2\vdash 1}$ is shown in figure 7.2. The moves in on the right of the prearena correspond to M, while moves on the left correspond to Γ.

Reduction from $\mathrm{RML}^{\text{P-Str}}_{2\vdash 1}$. The reduction is inductive on the construction of the canonical form. We make the construction indexed by initial moves, with each automaton $\mathcal{A}_i$ recognising the appropriate language restricted to the initial move i. The construction to combine these into one automaton as per the specification above is a straightforward union of the automata and merging of the initial states.

Our inductive hypothesis is slightly stronger than that the constructed automata recognises the appropriate languages. We also require the following conditions on the automaton $\mathcal{A}_i^M$:

- Initial states are never revisited (or have data values assigned to them)
- The automaton is deterministic
- Each state can only ever "hold" data values of one, fixed, level. i.e. if configurations (q, f) and (q', f') are both reachable, and $f(d) = f'(d') \neq \bot$ then d and d' are of the same level.
- There is precisely one transition from the initial state, labelled $i, (0, \bot)$. We will call the target state of this transition the "secondary state" of the automaton. Further, this is the only transition in the automaton with signature $(0, \bot)$.
- If q and q' are (non-initial) final states in the automaton, then if there is a transition $(q, a, \bar{s}, p, \bar{p})$ then $(q', a, \bar{s}, p, \bar{p})$ is also a transition.

Recall we will be using data values to encode O-pointers, which are all data values are used for, with each question and answer sharing a data value, and a question having fresh data value whose predecessor is the data value of its justifying move. Also recall we will only be encoding O-pointers, as P-pointers are recoverable from the underlying sequence of moves.

In general, the data values will simply remember the state of the automaton they were in when last read, and this will be sufficient to restrict O's moves to valid ones. The exception to this is caused by our ability to update the ancestors of the currently-read data value. It will be helpful to bear in mind that in the following construction, this ability is only used by the $\mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}\, M$ case, or where inherited from constituent automata. i.e. we only use it to update the memory of shared variables.

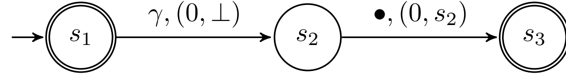
Notation describing NDCMA. In the following, transitions of NDCMA are described in a couple of ways. The most standard notation we use is to write something of the form $p \xrightarrow{m,(k,\bar{p})} q, \bar{q}$. Here we have $m \in \Sigma$, $k \in \mathbb{N}$, $p, q \in Q$, and $\bar{p}, \bar{q} \in (Q_\bot)^{k+1}$. This means that $(q, \bar{q}) \in \delta_k(p, m, \bar{p})$. Note that we have introduced an additional element $k \in \mathbb{N}$ to clarify the level of the data value being read by the transition. We may say a data value d is "at" a state q to mean $f(d) = q$.

Sometimes, we write $\binom{s}{\bar{s}}$ for the $k+1$-vector of elements of $Q_\perp$ obtained by putting s "on top" of the k-vector $\bar{s}$. Similarly $\binom{\bar{s}}{s}$ puts s "below" $\bar{s}$.

Sometimes in these transitions the final $\bar{q}$ is omitted: in this case it is implicitly assumed to only update the currently-read data value, which is updated to q. Formally: this means $\bar{q} = \bar{p}[q/p_k]$. When omitting this final $\bar{q}$, it is possible to draw the automaton in a relatively standard manner (e.g. in the first couple of cases below).

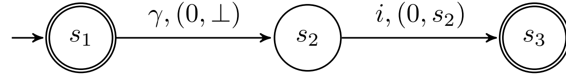
7.2.2.1 $()$: unit

For $[\![\Gamma \vdash () : \mathsf{unit}]\!]$ the complete plays of the strategy are of the form $\gamma \; \bullet$ (or the empty play). Hence $\mathcal{A}_\gamma$ is simply:



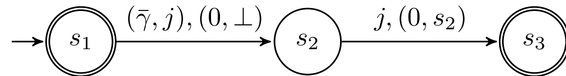
7.2.2.2 i : int

This is also straightforward, identical to the last case but with a differently labelled move:



7.2.2.3 $x^\beta : \beta$

Here we have $\Gamma \vdash x^\beta$, so $x : \beta$ is in Γ. Thus the initial moves have an x-component, so an initial move is of the form $(\bar{\gamma}, j)$ where j is in the x-component. For such an initial move, the plays recognised are just $\{\binom{(\bar{\gamma},j)}{d}\binom{j}{d} \; : \; d \in \mathcal{D} \text{ is level-0}\}$, and again the appropriate automaton is straightforwardly given:



7.2.2.4 succ(x^{int}) : int and pred(x^{int}) : int

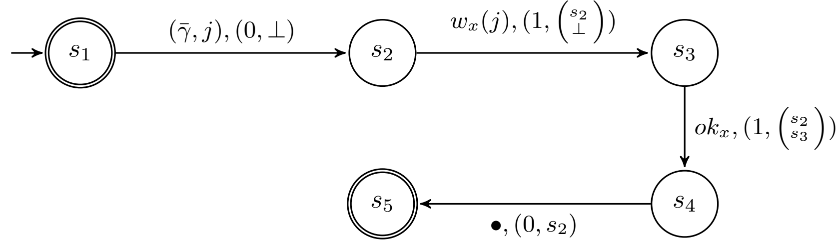
These are just as in the previous case, but adding or subtracting one to the j (modulo the fragment of $\mathbb{Z}$ being used).

7.2.2.5 $x^{\mathsf{int\ ref}} := y^{\mathsf{int}} : \mathsf{unit}$

Here we have $\Gamma \vdash x^{\mathsf{int\ ref}} := y^{\mathsf{int}}$, so $x : \mathsf{int\ ref}$ and $y : \mathsf{int}$ are in Γ. Thus the initial moves have a y-component, say j. Thus the language recognised by $\mathcal{A}_{(\bar{\gamma},j)}$ is just:

$$\{\binom{(\bar{\gamma},j)}{d}\binom{write_x(j)}{d'}\binom{ok_x}{d'}\binom{\bullet}{d} \mid d \text{ is level-0 and } pred(d') = d\}$$

This is recognised by the following automaton:



7.2.2.6 $!x^{\mathsf{int\ ref}} : \mathsf{int}$

This is similar to the previous case, except that the value for P to return is given by O's reponse to $read_x$. The desired language for $\mathcal{A}_\gamma$ is is just

$$\{(\gamma, d)(read_x, d')(j_x, d')(j, d) \mid j \in \mathbb{N}, d \text{ is level-0 and } pred(d') = d\}$$

This is recognised by a very similar automaton to the previous case, except that from state s_3 the automaton splits into different states for each possible answer j_x.

7.2.2.7 **if** x^β **then** M **else** $N : \theta$

The initial move contains an x-component. If this x-component is 0 then the automaton is as the as the automaton for N, otherwise it is as the automaton for M.

7.2.2.8 **mkvar**$(\lambda x^{\mathsf{unit}}.M, \lambda y^{\mathsf{int}}.N) : \mathsf{int\ ref}$

Here we have $\Gamma, x : \mathsf{unit} \vdash M : \mathsf{int}$ and $\Gamma, y : \mathsf{int} \vdash N : \mathsf{unit}$, and this "bad-variable" construction uses these methods as read- and write-methods respectively. The string projection of the language for $[\![\mathbf{mkvar}(\lambda x^{\mathsf{unit}}.M, \lambda y^{\mathsf{int}}.N)]\!]$ is then

$$\gamma \cdot \bullet \cdot (read \cdot L_M + \sum_j write(j) \cdot L_N^j)^*$$

Where L_M is the language for $[\![M]\!]$ without the initial move, and L_N^j is the language $[\![N]\!]$ when $y = j$, without the initial move.

For an initial move $\bar{\gamma}$, we make the following construction of $\mathcal{A}_{\bar{\gamma}}^{\mathbf{mkvar}(\lambda x.M, \lambda y.N)}$:

- The set of states is the disjoint union of the states of $\mathcal{A}^M_{\bar{\gamma}}$, and each $\mathcal{A}^N_{(\bar{\gamma},j)}$, minus the initial states, plus additional states (1), (2), and (3).
- The initial state is the state (1).
- The final states are those which are final in the constituent automata $\mathcal{A}^M_{\bar{\gamma}}$ and each $\mathcal{A}^N_{(\bar{\gamma},j)}$, and (1) and (3).
- The transition relation is given as follows:
  - $(1) \xrightarrow{\bar{\gamma},(0,\perp)} (2)$
  - $(2) \xrightarrow{\bullet,(0,(2))} (3)$
  - $(3) \xrightarrow{read,(1,\begin{pmatrix}(3)\\ \perp\end{pmatrix})} s_M$ where s_M is the secondary state of $\mathcal{A}^M_{\bar{\gamma}}$
  - $(3) \xrightarrow{write(j),(1,\begin{pmatrix}(3)\\ \perp\end{pmatrix})} s_{N,j}$ where $s_{N,j}$ is the secondary state of $\mathcal{A}^N_{(\bar{\gamma},j)}$
  - For all transitions $s_1 \xrightarrow{m,(0,\bar{\xi})} s_2$ in a constituent automaton which are not initial transitions in those automata, we have the transition $s_1 \xrightarrow{m,(1,\begin{pmatrix}(3)\\ \bar{\xi}\end{pmatrix})} s_2$.
  - For all other transitions $s_1 \xrightarrow{m,(i,\begin{pmatrix}\xi_0\\ \bar{\xi}\end{pmatrix})} s_2$ in a constituent automaton with $i \neq 0$, we have the transition $s_1 \xrightarrow{m,(i,\begin{pmatrix}(3)\\ \bar{\xi}\end{pmatrix})} s_2$. (Note that since there are no top-level local variables shared between M and N, these transitions cannot interfere with the level-0 data value.)
  - From each final state, s, in one of the constituent automata, we add transitions $s \xrightarrow{read,(1,\begin{pmatrix}(3)\\ \perp\end{pmatrix})} s_M$ and $s \xrightarrow{write(j),(1,\begin{pmatrix}(3)\\ \perp\end{pmatrix})} s_{N,j}$ (where s_M and $s_{N,j}$ are as before)

We note that determinism is inherited from the constituent automata. Further, the only transitions from final states we need to add for the inductive hypothesis have already been added.

7.2.2.9 while M do N

Here $M : \mathsf{int}$ and $N : \mathsf{unit}$. The strategy $[\![\mathbf{while}\, M\, \mathbf{do}\, N]\!]$ plays as if playing M until the final move would be made. If this would be 0, P gives the $\bullet$ answer to the initial move, and stops. Otherwise it plays as if playing N, until the final move would be made, when it starts as if playing M again.

The automaton $\mathcal{A}_\gamma^{\mathbf{while}\, M\, \mathbf{do}\, N}$ is thus given by:

- The set of states is given by the disjoint union of the set of states of $\mathcal{A}_\gamma^M$ and $\mathcal{A}_\gamma^N$, without the initial states, plus new states (1) and (2).
- The initial state is (1).
- The final states are (1) and (2).
- The transitions are given as follows:
  - (1) $\xrightarrow{\bar{\gamma},(0,\perp)} s_M$ where s_M is the secondary state of $\mathcal{A}_\gamma^M$.
  - if s' is a final state of $\mathcal{A}_\gamma^M$ and $s \xrightarrow{m,(0,\bar{\xi})} s'$ is a transition in $\mathcal{A}_\gamma^M$ (note that since M is of ground type, such moves must be of level 0), with $m \neq 0$, we have the transition $s \xrightarrow{\epsilon} s_N$ where s_N is the secondary state of $\mathcal{A}_\gamma^N$. (We can compress the silent transition ϵ out, since by determinism of the strategy this is the only transition from s in $\mathcal{A}_\gamma^M$, and it only affects the unique level 0 data value.)
  - if s' is a final state of $\mathcal{A}_\gamma^M$ and $s \xrightarrow{m,(0,\bar{\xi})} s'$ is a transition in $\mathcal{A}_\gamma^M$, with $m = 0$, we have the transition $s \xrightarrow{\bullet,(0,\bar{\xi})} (2)$
  - if s' is a final state of $\mathcal{A}_\gamma^N$ and $s \xrightarrow{m,(0,\bar{\xi})} s'$ is a transition in $\mathcal{A}_\gamma^N$, we have the transition $s \xrightarrow{\epsilon} s_M$ where s_M is the secondary state of $\mathcal{A}_\gamma^M$. (Again, we can compress the silent transition ϵ out, since by determinism of the strategy this is the only transition from s in $\mathcal{A}_\gamma^N$, so it can be "folded" into the transitions to s'.)
  - Internal transitions of the $\mathcal{A}^M$ and $\mathcal{A}^N$ automata are preserved.

Determinism is inherited from the constituent automata, and there are no transitions from final states that need be added. The only subtlety in this construction is preventing data values used in one run-through of $[\![M]\!]$ or $[\![N]\!]$ being used in subsequent

run-throughs, but this will already have been prevented in the construction of $\mathcal{A}^M$ and $\mathcal{A}^N$.

7.2.2.10 $\mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}\, M : \theta$

We assume we have a family of automata, $\mathcal{A}^M_\gamma$, recognising the strategy $[\![\Gamma, x : \mathsf{int\ ref} \vdash M : \theta]\!]$. $[\![\Gamma \vdash \mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}\, M : \theta]\!]$ is constructed by restricting behaviour of x to "good variable" behaviour (i.e. after a read-move the response is an immediate reply of the last integer written to the variable), and then hiding those moves. The automata construction is done in these two stages.

Restriction to good-variable behaviour. The value of the variable will be stored in both the current state of the automaton, and by the level-0 data value. The level-0 data value will be used to ensure that when O switches between threads (see the λ-abstraction construction in section 7.2.2.11), the correct variable value is retained. By keeping the value in the current state, the correct value is retained when moves in Γ are being made. Assume the finitary fragment we are using is $\{0, 1, \ldots, K\}$. Let Q be the non-initial states of $\mathcal{A}^M_\gamma$. We construct $\mathcal{C}_\gamma$ as follows:

- The states of the automaton are $\{q_I\} \uplus (Q \times \{0, 1, \ldots, K\})$
- The final states are q_I and those which are final in $\mathcal{A}^M_\gamma$ paired with any integer.
- The initial state is q_I
- The transitions are given as follows:
  - $q_I \xrightarrow{q_0,(0,\perp)} (s_M, 0)$ where s_M is the secondary state of $\mathcal{A}^M_\gamma$.
  - if $q_1 \xrightarrow{m,(k,\begin{pmatrix} s_0 \\ \vdots \\ s_k \end{pmatrix})} q_2, \begin{pmatrix} t_0 \\ \vdots \\ t_k \end{pmatrix}$ is in $\mathcal{A}^M_\gamma$ where m is not an *x-write* move or a response to an *x-read* move, then:
    - if m is a q_j move for some $j \neq 0$, for each $i_0, i_1, \ldots, i_k \in \{0, 1, \ldots, K\}$, we have the transition $(q_1, i_0) \xrightarrow{m,(k,\begin{pmatrix} (s_0,i_0) \\ \vdots \\ (s_k,i_k) \end{pmatrix})} (q_2, i_0), \begin{pmatrix} (t_0,i_0) \\ \vdots \\ (t_k,i_0) \end{pmatrix}$ (for convenience, if $s_k = \perp$ we interpret (s_k, i) as $\perp$ also).

* if m is not a q_j move, for each $i, j_0, j_1, \ldots, j_k \in \{0, 1, \ldots, K\}$, we have the transition $(q_1, i) \xrightarrow{m,(k,\begin{pmatrix}(s_0,j_0)\\ \vdots \\ (s_k,j_k)\end{pmatrix})} (q_2, i), \begin{pmatrix}(t_0,i)\\ \vdots \\ (t_k,i)\end{pmatrix}$.

- For each j, if $q_1 \xrightarrow{write_x(j),(1,\begin{pmatrix}s_0\\ \bot\end{pmatrix})} q_2, \begin{pmatrix}t_0\\ t_1\end{pmatrix}$ is in $\mathcal{A}_\gamma^M$, then we have $(q_1, i_1) \xrightarrow{write_x(j),(1,\begin{pmatrix}(s_0,i_0)\\ \bot\end{pmatrix})} (q_2, j), \begin{pmatrix}(t_0,j)\\ (t_1,j)\end{pmatrix}$ for each i_1, i_0.
- For each response to an x-read move, j_x, if $q_1 \xrightarrow{j_x,(1,\begin{pmatrix}s_0\\ s_1\end{pmatrix})} q_2, \begin{pmatrix}t_0\\ t_1\end{pmatrix}$ is in $\mathcal{A}_\gamma^M$, then we have $(q_1, j) \xrightarrow{j_x,(1,\begin{pmatrix}(s_0,i_0)\\ (s_1,i_1)\end{pmatrix})} (q_2, j), \begin{pmatrix}(t_0,j)\\ (t_1,j)\end{pmatrix}$ for each i_0, i_1.

Note that adding the transitions between accepting states as required by the inductive hypothesis will not change the language recognised, since all the outward transitions added would be labelled with a q_j move, and by construction these moves require a level-0 data value to be in the correct place.

Hiding $\mathcal{A}_\gamma^{\mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}\, M}$ is constructed from $\mathcal{C}_\gamma$ as follows:

If we are in a configuration (s_1, f) of $\mathcal{C}_i$ where we can perform a transition $s_1 \xrightarrow{m_x,(j,\bar{s})} s_2, \bar{t}$ where m_x is an x-move then by determinism of strategies combined with the restriction to good variable behaviour, it is the only possible transition from this configuration. Thus for every state s_0 of $\mathcal{C}_\gamma$ and every possible "signature" $\begin{pmatrix}t_0\\ t_0'\end{pmatrix}$, there is a unique maximal (and not necessarily finite) sequence of transitions:

$$s_0 \xrightarrow{m_0,(1,\begin{pmatrix}t_0\\ \bot\end{pmatrix})} s_1, \begin{pmatrix}t_1\\ t_1'\end{pmatrix} \xrightarrow{m_1,(1,\begin{pmatrix}t_1\\ t_1'\end{pmatrix})} s_2, \begin{pmatrix}t_2\\ t_2'\end{pmatrix} \xrightarrow{m_2,(1,\begin{pmatrix}t_2\\ \bot\end{pmatrix})} \ldots$$

or

$$s_0 \xrightarrow{m_0,(1,\begin{pmatrix}t_0\\ t_0'\end{pmatrix})} s_1, \begin{pmatrix}t_1\\ t_1'\end{pmatrix} \xrightarrow{m_1,(1,\begin{pmatrix}t_1\\ \bot\end{pmatrix})} s_2, \begin{pmatrix}t_2\\ t_2'\end{pmatrix} \xrightarrow{m_2,(1,\begin{pmatrix}t_2\\ t_2'\end{pmatrix})} \ldots$$

where each m_i is an x-move.

From each $\mathcal{C}_\gamma$ we construct the automaton $\mathcal{A}_\gamma^{\mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}\, M}$ by considering where this sequence terminates for each state. Everything is the same as in $\mathcal{C}_\gamma$ except for the transition relation, which is altered as follows:

- If the maximal sequence of x-moves with signature $\binom{t_0}{t'_0}$ out of state s_0 is empty then all transitions requiring signature $\binom{t_0}{t'_0}$ out of s_0 are unchanged.

- If the maximal sequence out of s_0 with signature $\binom{t_0}{t'_0}$ is finite and non-empty and ends in state s_n and with signature $\binom{t_n}{t'_n}$, then for every transition $s_n \xrightarrow{m,(1,\binom{t_n}{t'_n})} s_{n+1}, \bar{t}$ we add the transition $s_0 \xrightarrow{\epsilon} s_{n+1}$ (note that by determinism of the strategy and restriction to good variable behaviour, this ϵ transition can be compressed out without loss of determinism – since the moves in the sequence are all x-moves, the data values affected are minimal and can be inferred for the compression).

- All transitions on x-moves are removed

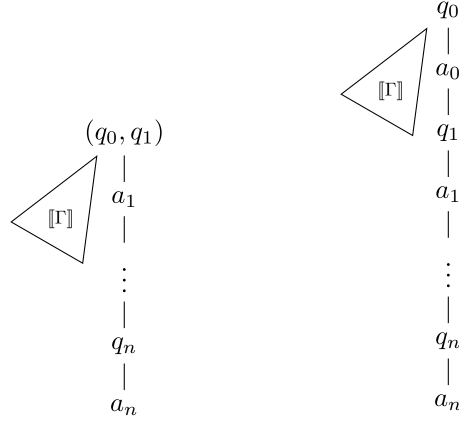
- Transitions from final states as required by the inductive hypothesis are added. This does not affect the language recognised, since the added transitions will require a level-0 data value to be "in" the relevant copy of Q_0, and there can only be one level-0 data value in runs of this automaton.

Determinism of the resulting automaton is inherited from determinism of $\mathcal{C}_\gamma$ (and thence from $\mathcal{A}^M_\gamma$).

7.2.2.11 $\lambda x^\beta.M : \beta \to \theta$

We have $\Gamma, x : \beta \vdash M : \theta$, and therefore assume there is a family of automata $\mathcal{A}^M_{\gamma,i}$ recognising $[\![M]\!]$. The prearenas for $[\![\Gamma, x \vdash M]\!]$ and $[\![\Gamma \vdash \lambda x.M]\!]$ are shown in Fig. 7.3. Note that the initial moves in $[\![\Gamma, x \vdash M]\!]$ contain an x-component, so may be considered pairs (γ, i_x), while the initial moves in $[\![\Gamma \vdash \lambda x.M]\!]$ contain the same Γ-component, but no x-component. The move q_0 therefore corresponds to the Γ-component, and the move q_1 precisely corresponds to an x-move.

$[\![\Gamma \vdash \lambda x.M]\!]$ is as follows: after an initial move γ, P plays the unique a_0-move $\bullet$, and waits for a q_1-move. Once O plays a q_1-move i_x, P plays as in $[\![\Gamma, x \vdash M]\!]$ when given an initial move (γ, i_x). However, as the q_1-moves are not initial, it is possible that O will play another q_1-move, i'_x. Each time O does this it opens a new thread which P plays as per $[\![\Gamma, x \vdash M]\!]$ when given initial move (γ, i'_x). Only O may switch



$[\![\Gamma, \beta \vdash \theta]\!]$ $\qquad$ $[\![\Gamma \vdash \beta \to \theta]\!]$

Figure 7.3: Prearenas for $[\![\Gamma, x : \beta \vdash M : \theta]\!]$ and $[\![\Gamma \vdash \lambda x^{\beta}.M : \beta \to \theta]\!]$

between threads, and this can only happen immediately after P plays an a_i-move (for any i). Hence we construct $\mathcal{A}_{\gamma}^{\lambda x.M}$ as follows:

- The set of states is the disjoint union of the set of non-initial states of each $\mathcal{A}^{M}_{(\gamma, i_x)}$, plus new states (1), (2), and (3).
- The initial state is (1)
- The final states are those that are final in each $\mathcal{A}^{M}_{(\gamma, i_x)}$, as well as (1) and (3).
- The transition relation is as follows:
  - $(1) \xrightarrow{\gamma,(0,\perp)} (2)$
  - $(2) \xrightarrow{a_0,(0,(2))} (3)$
  - For each i_x, $(3) \xrightarrow{i_x,(1,\left(\begin{smallmatrix}(3)\\ \perp\end{smallmatrix}\right))} s_{i_x}$ where s_{i_x} is the secondary state of $\mathcal{A}^{M}_{(\gamma, i_x)}$
  - If $s_1 \xrightarrow{m,(j,\bar{s})} s_2, \bar{t}$ is a (non-initial) transition in one of the $\mathcal{A}^{M}_{(\gamma, i_x)}$, then:
    * if m is a q_i or a_i move, $s_1 \xrightarrow{m,(j+1,\left(\begin{smallmatrix}(3)\\ \bar{s}\end{smallmatrix}\right))} s_2, \left(\begin{smallmatrix}(3)\\ \bar{t}\end{smallmatrix}\right)$ is a transition.
    * if m is a move in $[\![\Gamma]\!]$, $s_1 \xrightarrow{m,(j,\bar{s}[(3)/s_0])} s_2, \bar{t}[(3)/t_0]$ is a transition.

- If s_1 and s_1' are both (non-initial) final states and $s_1 \xrightarrow{m,(j,\bar{s})} s_2, \bar{t}$ is a transition already given by the above rules, then $s_1' \xrightarrow{m,(j,\bar{s})} s_2, \bar{t}$. (This allows O to switch between threads).

7.2.2.12 $\mathbf{let}\, x^\beta = M \,\mathbf{in}\, N : \theta$

Here we have $\Gamma \vdash M : \beta$ and $\Gamma, x : \beta \vdash N : \theta$. The initial moves of $[\![\Gamma, x : \beta \vdash N : \theta]\!]$ contain an x-component, so we index the family of automata recognising $[\![\Gamma, x : \beta \vdash N : \theta]\!]$ as $\mathcal{A}^N_{\gamma,j}$ where j is the x-component. The family of automata recognising $[\![M]\!]$ are indexed as $\mathcal{A}^M_\gamma$.

The strategy $[\![\mathbf{let}\, x^\beta = M \,\mathbf{in}\, N]\!]$ is essentially a concatenation of the strategies for $[\![M]\!]$ and $[\![N]\!]$, with the result of the $[\![M]\!]$ strategy determining the x-component of the initial move of $[\![N]\!]$. $\mathcal{A}_i^{\mathbf{let}\, x^\beta = M \,\mathbf{in}\, N}$ is constructed as follows:

If $\mathcal{L}(\mathcal{A}^M_i) = \{\binom{\gamma}{d}\binom{j}{d}\}$ then $\mathcal{A}_\gamma^{\mathbf{let}\, x^\beta = M \,\mathbf{in}\, N} = \mathcal{A}^N_{\gamma,j}$. Otherwise, by determinism of the strategy there cannot be a transition from the secondary state to a final state in $\mathcal{A}^M_\gamma$, and $\mathcal{A}_\gamma^{\mathbf{let}\, x^\beta = M \,\mathbf{in}\, N}$ is given by:

- The set of states is the disjoint union of the non-initial states of $\mathcal{A}^M_i$ and each $\mathcal{A}^N_{i,j}$, plus new a state (1).
- The initial state is (1).
- The final states are those which are final in each $\mathcal{A}^N_{\gamma,j}$, and (1).
- The transitions are given as follows:
  - $(1) \xrightarrow{\gamma,(0,\bot)} s_M$ where s_M is the secondary state of $\mathcal{A}^M_\gamma$
  - All transitions in $\mathcal{A}^M_\gamma$ not going to a final state (or from the initial state) are preserved
  - If $s_1 \xrightarrow{j,(0,s_3)} s_2, t$ is a transition in $\mathcal{A}^M_\gamma$ with s_2 final (in $\mathcal{A}^M_\gamma$) and $s_{N,j}$ is the secondary state of $\mathcal{A}^N_{\gamma,j}$:
    * if $s_{N,j} \xrightarrow{m,(0,s_{N,j})} s_4, t'$ is in $\mathcal{A}^N_{\gamma,j}$ then we have the transition $s_1 \xrightarrow{m,(0,s_3)} s_4, t'$.
    * if $s_{N,j} \xrightarrow{m,(1,\binom{s_{N,j}}{\bot})} s_4, \bar{t'}$ is in $\mathcal{A}^N_{\gamma,j}$ then we have the transition $s_1 \xrightarrow{m,(1,\binom{s_3}{\bot})} s_4, \bar{t'}$.

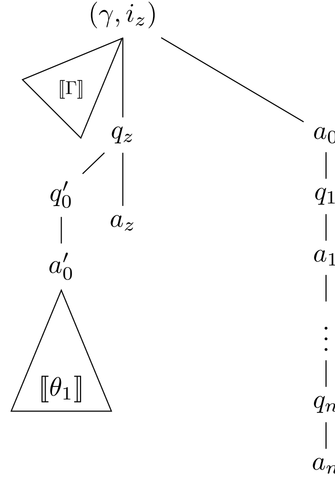


Figure 7.4: Prearena for $[\![\Gamma, z : (\beta \to \theta_1) \to \beta \vdash \theta]\!]$

– All other transitions in each $\mathcal{A}^N_{\gamma,j}$ are preserved unchanged.

– Transitions from final states as required by the inductive hypothesis are added. This does not affect the language recognised, since the added transitions will require a level-0 data value to be "in" the relevant copy of $\mathcal{A}^N_{\gamma,j}$, and there can only be one level-0 data value in runs of this automaton.

Determinism is inherited from $\mathcal{A}^M_\gamma$ and $\mathcal{A}^N_{\gamma,j}$.

7.2.2.13 $\mathbf{let}\, x = zy^\beta \,\mathbf{in}\, M : \theta$

As x must be of type β for this to be in $\mathrm{RML}^{\text{P-Str}}_{2\vdash 1}$, this is essentially the same as the previous case.

7.2.2.14 $\mathbf{let}\, x = z(\lambda y.M) \,\mathbf{in}\, N : \theta$

Here we have $\Gamma, y : \beta, z : (\beta \to \theta_1) \to \beta \vdash M : \theta_1$ and $\Gamma, x : \beta, z : (\beta \to \theta_1) \to \beta \vdash N : \theta$. As in the previous cases, plays in $[\![\mathbf{let}\, x = z(\lambda y.M) \,\mathbf{in}\, N]\!]$ consist of P playing $[\![z(\lambda y.M)]\!]$ until x has been evaluated, and then playing as N with this value of x. The prearena for this case is shown in figure 7.4.

Plays in $[\![\mathbf{let}\, x = z(\lambda y.M) \,\mathbf{in}\, N]\!]$ start with P playing q_z. O can then either play q'_0, starting an $[\![\lambda y.M]\!]$-thread, or play a_z, giving a value for x in the rest of the play. If O chooses the former, that thread is played as $[\![\lambda y.M]\!]$, with q'_0-moves providing a new value for y, until O plays an a_z move. Once O does play an a_z move, P plays as $[\![N]\!]$ with the answer O provided as the value for x.

In this construction a similar construction to that in the lambda abstraction case above will be used, to allow O to interleave plays of $[\![M]\!]$. At any point when O would be able to change threads, it is also able to finish evaluating M and give a value for x. Once this happens play continues in the corresponding $\mathcal{A}^N_{\gamma,j_x}$. The formal construction for $\mathcal{A}^{\mathbf{let}\,x=z(\lambda y.M)\,\mathbf{in}\,N}_{\gamma,i_z}$ is as follows:

- The set of states consists of:
  - Fresh states (1), (2), and (3)
  - A copy of the non-initial states of each $\mathcal{A}^M_{\gamma,i_y}$
  - A copy of the non-initial states of each $\mathcal{A}^N_{\gamma,j_x}$
- The initial state is (1)
- The final states are those which are final in each $\mathcal{A}^N_{\gamma,\mathrm{J}x}$, and (1)
- The transitions are:
  - $(1) \xrightarrow{\gamma,(0,\perp)} (2)$
  - $(2) \xrightarrow{q_z,(1,\begin{pmatrix}(2)\\ \perp\end{pmatrix})} (3)$
  - For each available a_z move labelled j_x we have the transition $(3) \xrightarrow{j_x,\begin{pmatrix}(2)\\(3)\end{pmatrix}} s_{N,j}$ where $s_{N,j}$ is the secondary state of $\mathcal{A}^N_{\gamma,j_x}$
  - For each available q_0' move labelled i_y, we have $(3) \xrightarrow{i_y,(2,\begin{pmatrix}(2)\\(3)\\ \perp\end{pmatrix})} s_M$ where s_M is the secondary state of $\mathcal{A}^M_{\gamma,i_y}$
  - If $s_1 \xrightarrow{m,(j,\bar{s})} s_2, \bar{t}$ is a (non-initial) transition in one of the $\mathcal{A}^M_{i_y}$, then:
    * if m is a q_i or a_i move in $[\![\theta_1]\!]$, $s_1 \xrightarrow{m,(j+2,\begin{pmatrix}(2)\\(3)\\ \bar{s}\end{pmatrix})} s_2, \begin{pmatrix}(2)\\(3)\\ \bar{t}\end{pmatrix}$ is a transition.
    * if m is a move in $[\![\Gamma]\!]$, $s_1 \xrightarrow{m,(j,\bar{s}[(2)/s_0])} s_2, \bar{t}[(2)/t_0]$ is a transition.
  - If $q_1 \xrightarrow{m,(k,\bar{s})} q_2, \bar{t}$ is a transition already defined by one of these, and q_1 is either (3) or a (non-initial) final state in one of the $\mathcal{A}^M_{\gamma,i_y}$, and q_3 is a (non-initial) final state in one of the $\mathcal{A}^M_{\gamma,i_y}$ then we have the transition $q_3 \xrightarrow{m,(k,\bar{s})} q_2, \bar{t}$.

– For each transition $s_{N,j} \xrightarrow{m,(k,\bar{s})} q, \bar{t}$ in each $\mathcal{A}^N_{\gamma,j_x}$, where $s_{N,j}$ is the secondary state of $\mathcal{A}^N_{\gamma,j_x}$, we have the transition $s_{N,j} \xrightarrow{m,(k,\bar{s}[(2)/s_0])} q, \bar{t}$

– All other transitions in each $\mathcal{A}^N_{\gamma,j_x}$ are left unchanged

– Transitions from final states as required by the inductive hypothesis are added. This does not affect the language recognised, since the added transitions will require a level-0 data value to be "in" the relevant copy of $\mathcal{A}^N_{\gamma,j_x}$, and there can only be one level-0 data value in runs of this automaton.

Determinism is inherited from the constituent automata.

7.2.2.15 $\mathbf{let}\, x = z\mathbf{mkvar}(\lambda u^{\mathsf{unit}}.M_1, \lambda v^{\mathsf{int}}.M_2)\,\mathbf{in}\, N : \theta$

This is very similar to the previous case: the difference is that the q_0' moves from the last case can now be either *read* or *write*(j), leading to playing as either $[\![M_1]\!]$ or $[\![M_2]\!]$ respectively. The formal construction is almost identical to that given above. □

This completes the proof of Theorem 7.2.2, and since equivalence of deterministic weak NDCMA is decidable (see Chapter 4) we get the result that observational equivalence of $\mathrm{RML}^{\text{P-Str}}_{2\vdash 1}$ is decidable:

Corollary 7.2.3. *Observational equivalence is decidable (with non-primitive recursive complexity) at types* $(\beta \to \cdots \to \beta) \to \beta \vdash \beta \to \cdots \to \beta$.

7.3 $\mathrm{RML}^{\mathrm{LHS-O-str}}_{2\vdash 1}$

It is important, for the reduction to NDCMA from $\mathrm{RML}^{\text{P-Str}}_{2\vdash 1}$, that variables cannot be partially evaluated: in prearenas where variables have only one argument, once a variable is evaluated those moves cannot be used to justify any future moves. If we could later return to them we would need ensure that they were accessed only in ways which did not break visibility. We now show that this can be done, using a slightly different encoding of pointers, for a fragment in which variables have unlimited arity, but each argument for the variable must be evaluated all at once. This means that the variables have their O-moves uniquely determined by the underlying sequence of moves.

This fragment strictly extends $\mathrm{RML}_{\mathrm{CMA}}$, keeping the permitted free variables the same, but extending the terms to all first-order types.

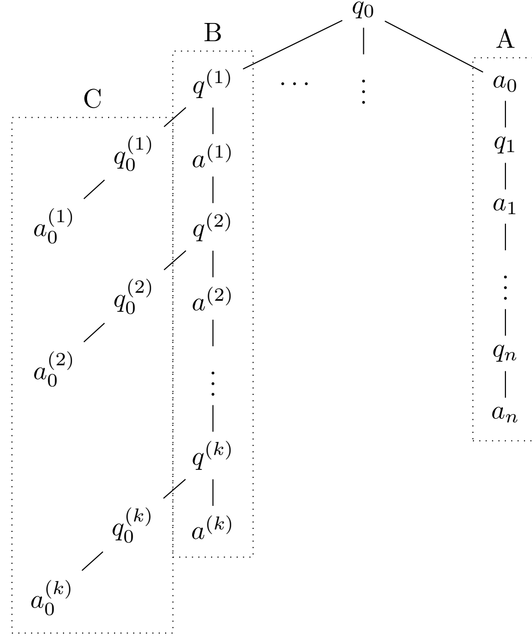


Figure 7.5: Shape of prearenas in $\mathrm{RML}_{2\vdash 1}^{\mathrm{LHS-O-str}}$

7.3.1 Fragment definition

Definition 7.3.1. The fragment we consider in this section, $\mathrm{RML}_{2\vdash 1}^{\mathrm{LHS-O-str}}$, consists of typed terms of the form $x_1 : \Theta_2^1, \ldots, x_n : \Theta_2^1 \vdash M : \Theta_1$ where the type classes Θ_i are as described below:

$$\Theta_0 ::= \mathsf{unit} \mid \mathsf{int} \qquad \Theta_1^1 ::= \Theta_0 \mid \Theta_0 \to \Theta_0 \mid \mathsf{int\ ref}$$

$$\Theta_1 ::= \Theta_0 \mid \Theta_0 \to \Theta_1 \mid \mathsf{int\ ref} \qquad \Theta_2^1 ::= \Theta_1 \mid \Theta_1^1 \to \Theta_2^1$$

This definition means we allow types of the form

$$(\beta \to \beta) \to \cdots \to (\beta \to \beta) \to \beta \vdash \beta \to \cdots \to \beta$$

where $\beta \in \{\mathsf{unit}, \mathsf{int}\}$. The shape of the prearenas for this fragment is shown in Figure Fig. 7.5.

Note that moves in section A of the prearena (marked in Figure 7.5) relate to the type Θ_1 on the RHS of the typing judgement, and that we need only represent O-pointers for this section, since the P-moves are all answers so have their pointers uniquely determined by well-bracketing. Moves in sections B and C of the prearena correspond to the types on the LHS of the typing judgement. Moves in section B need only have their P-pointers represented, since the O-moves are all answer moves.

Moves in section C have both their O- and P-pointers represented by the underlying sequence of moves: the P-pointers because all P-moves in this section are answer moves, the O-pointers by the visibility condition. We prove this below.

Proof. If a play $s\ \ q^{(i)}\ \ t\ \ q_0^{(i)}$ (with $q_0^{(i)}$ pointing to $q^{(i)}$) is a valid play on the arena above then we claim there is precisely one $q^{(i)}$ move in $view(s\, q^{(i)}\, t)$. More generally, we claim an O-view cannot have more than one $q^{(i)}$ move in it. This can be shown by induction on i. If $i = 1$ the claim is straightforward, as if $q^{(1)}$ is in the view then the preceding move in the view must be the unique q_0. For the inductive step, suppose otherwise, then by the inductive hypothesis there is at most one $q^{(i-1)}$ in the view, so also at most one $a^{(i-1)}$. Thus if there are two $q^{(i)}$ moves in the view, they must both point to this instance of $a^{(i-1)}$, but then the preceding move of each $q^{(i)}$ in the view must be that $a^{(i-1)}$. □

7.3.2 Deciding Observational Equivalence

As in the P-Strict case, we reduce $\mathrm{RML}_{2\vdash 1}^{\mathrm{LHS-O-str}}$ terms to deterministic weak nested data class memory automata that use data values to encode O-pointers. However, this time we do not need to represent any O-pointers on the LHS of the typing judgement, so we use data values only to represent pointers of the questions on the RHS. We do, though, need to represent P-pointers of moves on the LHS. This we do using the same "tagging" technique used for representing P-pointers in $\mathrm{RML}_{\text{O-Str}}$ [41] – which we recapped in Section 6.3.1.2. Hence for a term $[\![\Gamma \vdash M : \theta]\!]$ the data language we seek to recognise, $L([\![\Gamma \vdash M]\!])$, represents O-pointers in the following manner:

- The initial move takes the (unique) level-0 data value;
- Moves in $[\![\Gamma]\!]$ (i.e. in section B or C of the prearena shown in Fig. 7.5) take the data value of the previous move;
- Answer moves in $[\![\theta]\!]$ (i.e. in section A of the prearena) take the data value of the question they are answering; and
- Non-initial question moves in $[\![\theta]\!]$ (i.e. in section A of the prearena) take a fresh data value nested under the data value of the justifying answer move.

Note that we have to use different mechanisms for encoding P- and O-pointers, as each mechanism is unsuitable for encoding the other sort of pointer. The tagging mechanic used for P-pointers only works because we know the result strategy must be deterministic and so cannot be used for O-pointers. In contrast, the use of data-values is ideal for O-pointers, where O has many options for its pointers, and the word must uniquely represent which play is being represented, but data values would not allow us to restrict P's plays to deterministic strategies.

Theorem 7.3.1. *For every typed term $\Gamma \vdash M : \theta$ in $\mathrm{RML}^{\mathrm{LHS-O-str}}_{2\vdash 1}$ that is in canonical form we can effectively construct a deterministic weak nested data class memory automaton, $\mathcal{A}_M$, recognising the complete plays of $L(\llbracket \Gamma \vdash M \rrbracket)$.*

Proof. The reduction is inductive on the construction of the canonical form. We make the construction indexed by initial moves, with each automaton $\mathcal{A}_i$ recognising the appropriate language restricted to the initial move i. The construction to combine these into one automaton as per the specification above is a straightforward union of the automata and merging of the initial states.

Our inductive hypothesis is slightly stronger than that the constructed automaton recognises the appropriate language. We also require the following conditions on the automaton $\mathcal{A}_i^M$:

- Initial states are never revisited (or have data values assigned to them)
- The automaton is deterministic
- Each state can only ever "hold" data values of one, fixed, level.
- There is precisely one transition from the initial state, labelled $i, (0, \perp)$. We will call the target state of this transition the "secondary state" of the automaton. Further, this is the only transition in the automaton with signature $(0, \perp)$.
- If q and q' are (non-initial) final states in the automaton, then if there is a transition (q, a, ξ, p, ξ') then (q', a, ξ, p, ξ') is also a transition.

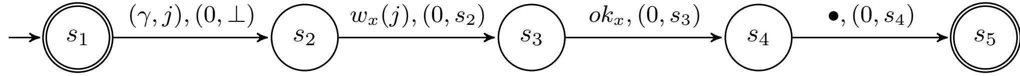
Note that in these constructions, in encoding P-pointers, we will initially generate automata that can recognise words encoding multiple P-pointers, contra to the encoding of P-pointers described in Section 6.3.1.2. However, it is straightforward

to afterwards intersect the resulting automaton with an automaton that restricts the construction to have at most one p-source and one p-target move. We therefore omit this detail from the following constructions.

For the cases $()$: unit, i : int, x^{β} : β, $\mathbf{succ}(x^{\mathsf{int}})$: int, and $\mathbf{pred}(x^{\mathsf{int}})$: int the languages, and hence the constructions, are exactly as in $\mathrm{RML}_{2\vdash 1}^{\text{P-Str}}$. Although the remaining cases are very similar to the $\mathrm{RML}_{2\vdash 1}^{\text{P-Str}}$ constructions, the languages differ in their use of data values. We deal with the remaining cases here:

7.3.2.1 $x^{\mathsf{int\ ref}} := y^{\mathsf{int}}$: unit

Here we have $\Gamma \vdash x^{\mathsf{int\ ref}} := y^{\mathsf{int}}$, so x : $\mathsf{int\ ref}$ and y : int are in Γ. Thus the initial moves have a y-component, say j. Thus the language recognised by $\mathcal{A}_{(\gamma,j)}$ is just $\{\binom{(\gamma,j)}{d}\binom{write_x(j)}{d}\binom{ok_x}{d}\binom{\bullet}{d} \mid d, \in \mathcal{D}$ and d is level-0$\}$. This is recognised by the following automaton:



7.3.2.2 $!x^{\mathsf{int\ ref}}$: int

This is similar to the previous case, only the value to return is given by O's play in the x-section of Γ. The language recognised by $\mathcal{A}_{\gamma}$ is just:

$$\{(\gamma, d)(read_x, d)(j_x, d)(j, d) \mid d \in \mathcal{D} \text{ and } d \text{ is level-0}\}$$

The automaton is thus similar to that given above, except that from state s_3 the automaton splits into different states for each possible answer j_x.

7.3.2.3 **if** x^{β} **then** M **else** N : θ

The initial move contains an x-component. If this x-component is 0 then the automaton is as the as the automaton for N, otherwise it is as the automaton for M.

7.3.2.4 $\mathbf{mkvar}(\lambda x^{\mathsf{unit}}.M, \lambda y^{\mathsf{int}}.N)$: $\mathsf{int\ ref}$

This is exactly the same as in the $\mathrm{RML}_{2\vdash 1}^{\text{P-Str}}$-case, except that now the constituent automata must be level 0. The same construction works.

7.3.2.5 **while** M **do** N : unit

Again, the same construction as that used in the $\mathrm{RML}_{2\vdash 1}^{\text{P-Str}}$ case works.

7.3.2.6 $\mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}\, M : \theta$

This is similar to the construction for $\mathrm{RML}_{2\vdash 1}^{\text{P-Str}}$, but this time the value of the variable will be stored just by the level-0 data value. This will correctly capture the scope of the variable. (Keeping the variable value in automaton state as well as by the level-0 data value is no longer necessary since the plays in $[\![\Gamma]\!]$, unlike in the $\mathrm{RML}_{2\vdash 1}^{\text{P-Str}}$ case, will use the same data value as the most recent move in $[\![M]\!]$.)

Since the construction is another straightforward restriction to good-variable behaviour and then hiding of the x-moves, we omit the details.

7.3.2.7 $\lambda x^{\beta}.M : \beta \to \theta$

This case is almost identical to the corresponding case in the $\mathrm{RML}_{2\vdash 1}^{\text{P-Str}}$ constructions: again the automaton recognises plays of the form $\overset{\frown}{q_0\ a_0}$ followed by interleavings of plays of $[\![M]\!]$. The only difference in the construction is that in this case, since data values are used differently, moves in $[\![\Gamma]\!]$ do not need to be handled differently to q_i and a_i moves. This is a straightforward alteration and we omit the full description here.

7.3.2.8 $\mathbf{let}\, x^{\beta} = M\, \mathbf{in}\, N : \theta$

This is very similar to the equivalent case in $\mathrm{RML}_{2\vdash 1}^{\text{P-Str}}$: again it is a straightforward concatenation of the languages for $\mathcal{A}^M$ and $\mathcal{A}^N$, with the outcome of $\mathcal{A}^M$ determining which copy of $\mathcal{A}^N$ is used.

7.3.2.9 $\mathbf{let}\, x = zy^{\beta}\, \mathbf{in}\, M : \theta$

We have $\Gamma, x : \theta', z : \beta \to \theta', y : \beta \vdash M : \theta$. Plays in $[\![\mathbf{let}\, x = zy^{\beta}\, \mathbf{in}\, M]\!]$ begin with P copying the y-component of the initial move into the z-component, and O must respond with the unique answer, $\bullet_z$ (which corresponds to the initial move of $[\![\theta']\!]$). Play then continues as $[\![M]\!]$ except that all x-moves are relabelled as z-moves, hereditarily justified by the occurrence of $\bullet_z$ O was forced to play. The pointers for moves justified by $\bullet_z$ will have to be made explicit as part of the construction.

$\mathcal{A}_{\gamma,i_y,i_z}^{\mathbf{let}\, x=zy^{\beta}\, \mathbf{in}\, M}$ is then constructed as follows:

- The states are two copies of the non-initial states of $\mathcal{A}_{\gamma,i_y,i_z,\bullet_x}^{M}$ (where $\bullet_x$ is the move $\bullet_z$ that O will be forced to play, relabelled as an x-move) plus new states

(1), (2) and (3). The second copy of $\mathcal{A}^M_{\gamma,i_y,i_z,\bullet_x}$ will be used to encode P-pointers, so we write state s in the second copy as $\overset{\bullet}{s}$.

- The initial state is (1).
- The final states are those final in either copy of $\mathcal{A}^M_{\gamma,i_y,i_z,\bullet_x}$, and (1).
- The transitions are as follows:
  - (1) $\xrightarrow{(\gamma,i_y,i_z),(0,\perp)}$ (2)
  - (2) $\xrightarrow{j_z,(0,(2))}$ (3) where j_z is the initial move for y copied into the z-component.
  - (3) $\xrightarrow{\bullet_z,(0,(3))}$ s_M and (3) $\xrightarrow{\overset{\bullet}{\bullet}_z,(0,(3))} \overset{\bullet}{s}_M$ where s_M is the secondary state of $\mathcal{A}^M_{\gamma,i_y,i_z,\bullet_x}$.
  - $s_1 \xrightarrow{m,(k,\bar{s})} s_2, \bar{t}$ is a transition in $\mathcal{A}^M_{\gamma,i_y,i_z,\bullet_x}$ and m is not an x-move, then we have the transitions $s_1 \xrightarrow{m,(k,\bar{s})} s_2, \bar{t}$ and $\overset{\bullet}{s_1} \xrightarrow{m,(k,\overset{\bullet}{\bar{s}})} \overset{\bullet}{s_2}, \overset{\bullet}{\bar{t}}$ (where $\overset{\bullet}{\bar{s}}$ replaces each element s of $\bar{s}$ with $\overset{\bullet}{s}$).
  - $s_1 \xrightarrow{m_x,(k,\bar{s})} s_2, \bar{t}$ is a transition in $\mathcal{A}^M_{\gamma,i_y,i_z,\bullet_x}$ and m_x is a non-initial x-move, then we have the transitions $s_1 \xrightarrow{m_z,(k,\bar{s})} s_2, \bar{t}$ and $\overset{\bullet}{s_1} \xrightarrow{m_z,(k,\overset{\bullet}{\bar{s}})} \overset{\bullet}{s_2}, \overset{\bullet}{\bar{t}}$, where m_z is the relabelling of m_x into the z-component.
  - $s_1 \xrightarrow{m_x,(k,\bar{s})} s_2, \bar{t}$ is a transition in $\mathcal{A}^M_{\gamma,i_y,i_z,\bullet_x}$ and m is the initial x-move, then we have the transitions $s_1 \xrightarrow{m_z,(k,\bar{s})} s_2, \bar{t}$ and $\overset{\bullet}{s_1} \xrightarrow{m_z,(k,\overset{\bullet}{\bar{s}})} \overset{\bullet}{s_2}, \overset{\bullet}{\bar{t}}$ and $\overset{\bullet}{s_1} \xrightarrow{\overset{\circ}{m}_z,(k,\overset{\bullet}{\bar{s}})} \overset{\bullet}{s_2}$ $, \overset{\bullet}{\bar{t}}$, where m_z is the relabelling of m_x into the z-component.
  - Transitions from final states as required by the inductive hypothesis are added. This does not affect the language recognised, since the added transitions will require a level-0 data value to be "in" the relevant location, and there can only be one level-0 data value in runs of this automaton.

Determinism is inherited from the constituent automaton.

7.3.2.10 **let** $x = z(\lambda y.M)$ **in** $N : \theta$

Here we have $\Gamma, y : \beta, z : (\beta \to \beta) \to \theta_1 \vdash M : \beta$ and $\Gamma, x : \theta_1, z : (\beta \to \beta) \to \theta_1 \vdash N : \theta$. The prearena is shown in Fig. 7.6

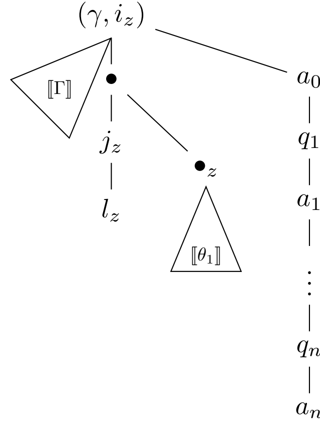


Figure 7.6: Prearena for $[\![\Gamma, y : \beta, z : (\beta \to \beta) \to \theta_1 \vdash N : \beta \to \cdots \to \beta]\!]$

Plays in $[\![\mathbf{let}\, x = z(\lambda y.M)\, \mathbf{in}\, N]\!]$ start with P playing $\bullet$. O can then either play j_z, starting an $[\![M]\!]$-thread, or play $\bullet_z$, the initial x-move. If O chooses the former, that thread is played to completion, as in $[\![M]\!]$. Once this is finished (with P playing the final move in $[\![\theta_1]\!]$), we return to the situation where O can play either j_z or $\bullet_z$. Once O does play $\bullet_z$, P plays as $[\![N]\!]$, except that all x-moves are renamed to z-moves (justified by $\bullet_z$). Further, whenever P plays in x (which becomes a z-move), O can again play j_z and start an $[\![M]\!]$ thread.

The automaton $\mathcal{A}_{\gamma,i_z}^{\mathbf{let}\, x=z(\lambda y.M)\, \mathbf{in}\, N}$ is constructed as follows:

- The set of states consists of:
  - Fresh states (1), (2), and (3)
  - Two copies of the set of non-initial states of $\mathcal{A}_{\gamma,\bullet_x,i_z}^{N}$, the second marked as $\overset{\bullet}{s}$
  - define $\mathcal{S}$, the set of states from which an $[\![M]\!]$-thread can be opened, as

    $\mathcal{S} = \{(3)\} \uplus \{r : (r = s \text{ or } r = \overset{\bullet}{s}) \text{ and } t \xrightarrow{m_x} s \text{ in } \mathcal{A}_{\gamma,\bullet_x,i_z}^{N} \text{ with } m_x \text{ a P-}x \text{ move}\}$

    We then take states (s, t) where s is a state in some $\mathcal{A}_{\gamma,i_y,i_z}^{M}$ and $t \in \mathcal{S}$
- The initial state is (1)
- The final states are those which are final in $\mathcal{A}_{\gamma,\bullet_x,i_z}^{N}$ (both tagged and untagged), and (1)

- The transitions are:
  - (1) $\xrightarrow{(\gamma,i_x),(0,\bot)}$ (2)
  - (2) $\xrightarrow{\bullet,(0,(2))}$ (3)
  - (3) $\xrightarrow{\bullet_z,(0,(3))}$ s_N and (3) $\xrightarrow{\dot{\bullet}_z,(0,(3))} \dot{s}_N$, where s_N is the secondary state of $\mathcal{A}^N_{\gamma,\bullet_x,i_z}$
  - If $s_1 \xrightarrow{m,(k,\bar{s})} s_2, \bar{t}$ is a transition in $\mathcal{A}^N_{\gamma,\bullet_x,i_z}$ and m is not an x-move, then we have the transitions $s_1 \xrightarrow{m,(k,\bar{s})} s_2, \bar{t}$ and $\dot{s}_1 \xrightarrow{m,(k,\dot{\bar{s}})} \dot{s}_2, \dot{\bar{t}}$
  - If $s_1 \xrightarrow{m_x,(k,\bar{s})} s_2, \bar{t}$ is a transition in $\mathcal{A}^N_{\gamma,\bullet_x,i_z}$ and m_x is a non-initial x-move, then we have the transitions $s_1 \xrightarrow{m_z,(k,\bar{s})} s_2, \bar{t}$ and $\dot{s}_1 \xrightarrow{m_z,(k,\dot{\bar{s}})} \dot{s}_2, \dot{\bar{t}}$, where m_z is the relabelling of m_x into the z-component.
  - If $s_1 \xrightarrow{m_x,(k,\bar{s})} s_2, \bar{t}$ is a transition in $\mathcal{A}^N_{\gamma,\bullet_x,i_z}$ and m is the initial x-move, then we have the transitions $s_1 \xrightarrow{m_z,(k,\bar{s})} s_2, \bar{t}$ and $\dot{s}_1 \xrightarrow{m_z,(k,\dot{\bar{s}})} \dot{s}_2, \dot{\bar{t}}$ and $\dot{s}_1 \xrightarrow{\mathring{m}_z,(k,\dot{\bar{s}})} \dot{s}_2$ $, \dot{\bar{t}}$, where m_z is the relabelling of m_x into the z-component.
  - If $s \in \mathcal{S}$ then for all transitions $t \xrightarrow{m_x,(k,\bar{t})} s, \bar{s}$ we have the transition $s \xrightarrow{j_z,(k,\bar{s})} (q_{M,j}, s), \bar{s}_{q_{M,j}}$, where $q_{M,j}$ is the secondary state of $\mathcal{A}^M_{\gamma,j_y,i_z}$, and $\bar{s}_{q_{M,j}}$ is the same as $\bar{s}$ but with the last element paired with $q_{M,j}$. Further:
    * If $p_1 \xrightarrow{m,(0,p'_1)} p_2, p'_2$ is in $\mathcal{A}^M_{\gamma,j_y,i_z}$ where p_2 is not final (in $\mathcal{A}^M_{\gamma,j_y,i_z}$), we have $(p_1, s) \xrightarrow{m,(k,\bar{s}_{p'_1})} (p_1, s), \bar{s}_{p'_2}$
    * If $p_1 \xrightarrow{l,(0,p'_1)} p_2, p'_2$ is in $\mathcal{A}^M_{\gamma,j_y,i_z}$ where p_2 is final (in $\mathcal{A}^M_{\gamma,j_y,i_z}$), we have $(p_1, s) \xrightarrow{m,(k,\bar{s}_{p'_1})} s, \bar{s}$
  - Transitions from final states as required by the IH are added. This does not affect the language recognised, since the added transitions will require a level-0 data value to be "in" the sub-automaton, and there can only be one level-0 data value in runs of this automaton.

Determinism is inherited from the constituent automata.

7.3.2.11 $\textbf{let}\, x = z\textbf{mkvar}(\lambda u^{\mathsf{unit}}.M_1, \lambda v^{\mathsf{int}}.M_2)\,\textbf{in}\, N : \theta$

As in the P-strict proof, this is very similar to the previous case, with O choosing whether to start an M_1-thread or M_2-thread. The construction is a straightforward alteration to the previous case. □

This completes the proof of Theorem 7.3.1, and so, analogously to the P-strict case, we get the key result:

Corollary 7.3.2. *Observational equivalence is decidable (with non-primitive recursive complexity) at types* $(\beta \to \beta) \to \cdots \to (\beta \to \beta) \to \beta \vdash \beta \to \cdots \to \beta$.

7.4 Summary

In this chapter we defined $\mathrm{RML}_{2\vdash 1}$, consisting of terms of order 1 with free variables of order at most 2. Prior to [18] (on which this chapter is based), the largest subfragment of $\mathrm{RML}_{2\vdash 1}$ that had been shown to have a decidable observational equivalence problem was $\mathrm{RML}_{\mathrm{CMA}}$, which further restricted terms to arity 2 and free variables' arguments to be arity at most 1. Observational equivalence of this subfragment was shown, in [40], to be reducible to emptiness of class memory automata.

We showed that by using the more expressible nested data class memory automata (NDCMA), we were able to decide observational equivalence for (certain) first-order terms of all arities. In particular, in Section 7.3 we defined $\mathrm{RML}_{2\vdash 1}^{\mathrm{LHS-O-str}}$ which extends $\mathrm{RML}_{\mathrm{CMA}}$ to all arities, and reduced these terms to deterministic weak NDCMA. We also idenfitied the P-strict subfragment of $\mathrm{RML}_{2\vdash 1}$, $\mathrm{RML}_{2\vdash 1}^{\text{P-Str}}$, which consists of first-order terms in which the free variables are of order at most 2 and arity at most 1, and showed that, by similar methods, terms in this fragment could be reduced to deterministic weak NDCMA.

It is natural to ask why we could not extend the approach used for $\mathrm{RML}_{2\vdash 1}^{\mathrm{LHS-O-str}}$ to wider types. Here the issue is that the nested structure of the data values is insufficient to capture the visibility issues involved in being able to partially evaluate – and then return to – LHS types of the kind we saw in the reduction from $\mathrm{RML}_{2\vdash 1}^{\text{P-Str}}$.

We note that although these reductions show $\mathrm{RML}_{2\vdash 1}^{\mathrm{LHS-O-str}}$ and $\mathrm{RML}_{2\vdash 1}^{\text{P-Str}}$ to have decidable observational equivalence problems, the complexity of these algorithms is

non-primitive recursive, since equivalence of deterministic weak NDCMA is not in PR.

Chapter 8

Hardness and Undecidability Arguments

In Chapter 7 we saw that observational equivalence of first-order RML terms with certain second-order free variables is decidable in non-primitive recursive time. Two natural questions arising out of this are: (i) is observational equivalence of first-order terms with any second-order free variables decidable? and (ii) can observational equivalence of these fragments be decided more efficiently?

Previous work ([41, 40]) found observational equivalence was decidable at some second-order types of arity 1, and undecidable at second-order types where there is a first-order argument that is not the final argument. This left, as an open question, (iii) is observational equivalence at second-order types of the form $\vdash \beta \to \cdots \to \beta \to (\beta \to \beta) \to \beta$ decidable?

In this chapter we prove hardness and undecidability results that answer question (i), mostly answer question (ii), and begin to answer question (iii). In Section 8.1 we show that VASS coverability can be reduced to observational equivalence at types $\vdash \beta \to \beta \to \beta$, and hence this fragment is ExpSpace-hard. We further extend this argument to show that the non-primitive recursive problem of coverability of reset VASS is reducible to observational equivalence at types $\vdash \beta \to \beta \to \beta \to \beta$, and thus we answer question (ii) above. We then go on to extend these arguments to reductions to reachability, in particular beginning to answer question (iii) by showing that the undecidable problem of reset VASS reachability is reducible to observational equivalence at type $\vdash \beta \to \beta \to (\beta \to \beta) \to \beta$. In Section 8.2 we turn our attention to question (i), and show that there is a second-order type such that observational

Figure 8.1: Prearena shape for $\vdash \beta \to \beta \to \beta$

equivalence of terms with free variables of this type is undecidable. We do this by using the visibility restrictions on plays to encode links between threads of the free variables, and thus encode queue machines, which are Turing-complete.

8.1 VASS-based reductions

Recall a VASS is a vector addition system with state, and a reset VASS (RVASS) is a VASS augmented with transitions that set counters to 0. We will consider their coverability and reachability problems (see Section 2.1 for a recap).

8.1.1 VASS Coverability to $\vdash \beta \to \beta \to \beta$

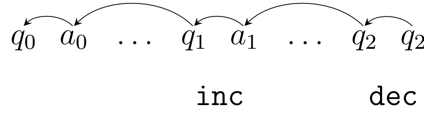
Recall in [40] it was shown that RML terms $\vdash M : \beta \to \beta \to \beta$ had a decidable observational equivalence problem via reduction to an exponentially larger class memory automata. In fact, as mentioned in Section 6.3.1.3, the reduction can be simplified to weak class memory automata, and hence observational equivalence for terms in canonical form is in 2-ExpSpace. Here we show an ExpSpace-hardness lower bound for this problem.

Theorem 8.1.1. *Given a VASS $\mathcal{A} = (Q, k, \Delta_i \cup \Delta_d, q_0)$ and target state $q_f \in Q$, there are* RML*-terms $\vdash M, M' : \mathsf{int} \to \mathsf{int} \to \mathsf{unit}$ such that $M \cong M'$ iff there is a run of $\mathcal{A}$ reaching state q_f.*

Proof. We will reason about the behaviour of the terms M and M' we construct using their game semantics. The shape of the arenas for these terms is shown in Fig. 8.1, and we note that the q_1 and q_2-moves correspond to the int type. Plays in these arenas

are of the form $q_0\ a_0\ q_1\ a_1\ (\ q_1\ a_1\ \mid\ q_2\ a_2\ )^*$ with each q_1 move justified by the unique a_0 move, and each q_2-move justified by any of the preceding a_1-moves.

The main difficulty in encoding a VASS in RML is storing arbitrarily large counter values. The key idea of this encoding is to use moves $q_1\ a_1$ to encode an increment to a counter, and corresponding $q_2\ a_2$ moves to make a decrement. A local variable in each $q_1\ a_1$-thread keeps track of whether that increment has been decremented, and when $q_2\ a_2$ is played, the term checks that the parent $q_1\ a_1$-thread hasn't already been decremented, then stores it as decremented. Pictorially:



Crucially, there is no limit to how many $q_1\ a_1$-threads can be made in the play, and they are all visible to O, and so the problem of arbitrarily large counter values is resolved.

In a little more detail, recall a VASS has a set of states, Q, an initial state $q_0 \in Q$, counters labelled 1, ..., d, a set of increment transitions Δ_i and a set of decrement transitions Δ_d. The play representing this VASS will store the current state in a global variable. To make a transition, O plays a q_1 or q_2 move, providing an input int. We treat this input as naming which transition is firing.

- In the case that it is a $\delta = (q, \bar{e_k}, q') \in \Delta_i$, O must have played a q_1 move. P will check that the current global state is q, and then update it to q'. In addition, this $q_1\ a_1$-thread will have a local variable storing the fact that this thread corresponds to an increment to counter k. It will have another local variable, storing whether this increment has been decremented yet.

- if $\delta = (q, -\bar{e_k}, q') \in \Delta_d$, O must have played a q_2 move. P will check the current state is q and update it to q' as above. Then it checks the local variables in the parent $q_1\ a_1$-thread to check that the parent $q_1\ a_1$-moves do in fact correspond to an increment for counter k, and that it has not yet been decremented. Finally it updates the local variable to mark the $q_1\ a_1$-thread as having been decremented.

```
1  let
2    State = ref [[q_0]]
3  in
4    λx^int.
5      let
6        Counter_Num = ref 0
7        Decremented = ref False
8      in
9        assert (x ∈ [[Δ_i]])
10       ⋃_{(q,ē_i,q')∈Δ_i} :
11         if x == [[(q, ē_i, q')]] then
12           assert (State == [[q]]); State := [[q']]; Counter_Num := i
13         else skip
14       assert* (!State ≠ [[q_f]])
15       λy^int.
16         assert (y ∈ [[Δ_d]])
17         assert (!Decremented == False)
18         ⋃_{(q,−ē_i,q')∈Δ_d} :
19           if y == [[(q, −ē_i, q')]] then
20             assert (State == [[q]]); assert (!Counter_Num == i);
21             State := [[q']]; Decremented := True
22           else skip
23         assert* (!State ≠ [[q_f]])
```

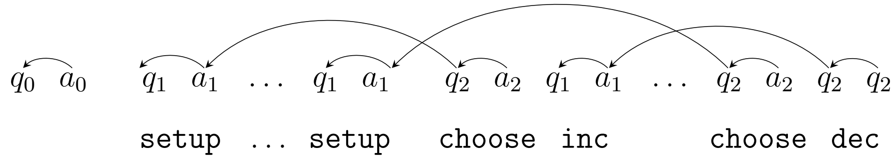
Figure 8.2: The term encoding VASS coverability

The term M is shown in Fig. 8.2, and we obtain M' by simply removing the starred assert statements. We assume the VASS to have states Q, increment transitions Δ_i and decrement transitions Δ_d, and counters labelled $1, \ldots, d$. Further, we match each transition in Δ_i and Δ_d to one of the possible values of the type int, using an injective function $[\![-]\!] : \Delta_i \cup \Delta_d \to \mathbb{N}_{fin}$. We similarly match states, using the function $[\![-]\!] : Q \to \mathbb{N}_{fin}$ (we use the same name for both functions: by context it will be obvious which is being used). We use the shorthand **assert** (C) for the term **if** C **then skip else** Ω. Sometimes the condition in these is of the form $a \in A$, where A is a finite set: this must obviously in fact be expressed using multiple if statements. Further, we use the symbol $\bigcup_{i \in I} : X_i$ to denote blocks of code X_i that are duplicated for each $i \in I$. □

We have thus shown that observational equivalence of types int → int → unit is ExpSpace-hard, but the question remains whether it is this hard at all types $\beta \to \beta \to \beta$. We now show that this is the case, and briefly describe how the above encoding can be altered to use only terms of type unit → unit → unit. The int types

in the above were used to allow O to choose which transition was to be fired at each point, allowing the term to model the non-determinism of VASS. Hence we need an alternative means for O to provide the non-deterministic choice of which transition to fire next. At types $\mathsf{unit} \to \mathsf{unit} \to \mathsf{unit}$ the only choice O can make is whether to make a q_1-move or a q_2-move, and if the latter which thread it is justified by. We will exploit the choice of justifier as a source of non-determinism to allow us to choose the next transition.

To do this the we require the plays to start with O generating $\widehat{q_1 \; a_1}$-thread for each possible transition, and each thread having a local variable storing which transition it corresponds to. Once this setup is complete, plays alternate between the increment/decrement operations in the initial encoding, and O playing a q_2-move to choose what transition is to be played next. This alternating discipline on the plays can be enforced using appropriate global variables. The plays will hence look something like this:



(We have omitted pointers from q_1-moves, as they must always point to the unique a_0-move.)

The term for this encoding is shown in Fig. 8.3. Again we assume a labelling function $[\![-]\!] : Q \to \mathbb{N}_{fin}$, and a transition labelling function $[\![-]\!] : \Delta_i \cup \Delta_d \to \{1, \ldots, n\}$. We briefly explain how the variables are used in this term. We have the following global variables:

- *State*. This variable is used to keep track of the current state, just as in the encoding in Fig. 8.2.
- *Setup_Done*. This tracks whether the initial setup of the $\widehat{q_1 \; a_1}$-threads corresponding to the transitions has yet been fully performed. Whilst this variable is false assertions in the term prevent O from playing any q_2-moves. Once the nth $\widehat{q_1 \; a_1}$-thread has been played, the term will set this variable to True.

- *Next_Transition*. This variable is used in setup to keep track of the number of $\overleftarrow{q_1\ a_1}$-threads, and store in each of those a different number from 1 to n. Once the nth $\overleftarrow{q_1\ a_1}$-thread is made, the term sets this variable to 0. In the rest of the play this variable stores which choice O makes for the next transition to be played. When this variable is 0 it means we are waiting for O to choose a transition (by playing an appropriate q_1-move). When nonzero it is taken to be the number of the next transition to be taken.

And we have the following local variables in each $\overleftarrow{q_1\ a_1}$-thread:

- *Transition_Num*. This is used only in the initial $\overleftarrow{q_1\ a_1}$-threads that are setup at the start of the run. It is used to store the label of the transition which that thread corresponds to.

- *Counter_Num* and *Decremented*. These are used exactly as in the encoding in Fig. 8.2, to store which of the counters this increment is for, and whether it has yet been decremented.

Thus we have shown:

Theorem 8.1.2. *Observational equivalence of terms of type* $\vdash \beta \to \beta \to \beta$ *is* EXPSPACE-*hard.*

8.1.2 RVASS coverability to $\vdash \beta \to \beta \to \beta \to \beta$

Although we were able to reduce observational equivalence of terms of type $\beta \to \beta \to \beta$ to weak class memory automata [40] (and hence get decidability within 2-EXPSPACE, as discussed in Section 6.3.1.3), for terms of type $\beta \to \beta \to \beta \to \beta$ we reduced observational equivalence to equivalence of weak nested data class memory automata, which we showed to be at least as hard as coverability of VASS with resets (RVASS). Here we show that this increase in complexity was necessary, by showing that observational equivalence of terms of this type is at least as hard as RVASS coverability.

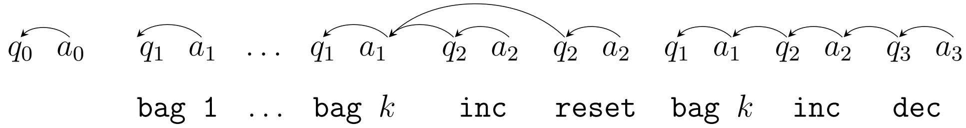
The key idea of this is to use $\overleftarrow{q_2\ a_2}$ and $\overleftarrow{q_3\ a_3}$-moves in the same way that $\overleftarrow{q_1\ a_1}$ and $\overleftarrow{q_2\ a_2}$-moves (resp.) were used in the encoding of VASS in Section 8.1.1, as increments and decrements. The "extra layer" of $\overleftarrow{q_1\ a_1}$-moves is then used to

```
 1  let
 2    State = ref ⟦q_0⟧
 3    Setup_Done = False
 4    Next_Transition = ref 1
 5  in
 6    λx^unit.
 7      let
 8        Transition_Num = ref 0
 9        Counter_Num = ref 0
10        Decremented = ref False
11      in
12        if (!Setup_Done == False) then
13          Transition_Num := Next_Transition;
14          if (!Next_Transition == n) then
15            Setup_Done := True; Next_Transition := 0
16          else Next_Transition++
17        else
18          assert (!Next_Transition ∈ ⟦Δ_i⟧)
19          ⋃_{(q,ē_i,q')∈Δ_i} :
20            if !Next_Transition == ⟦(q, ē_i, q')⟧ then
21              assert (State == ⟦q⟧); State := ⟦q'⟧;
22              Counter_Num := i; Next_Transition := 0
23            else skip
24          assert* (!State ≠ ⟦q_f⟧)
25          λy^unit.
26            assert (!Setup_Done == True)
27            if (!Next_Transition == 0) then
28              assert (!Transition_Num ≠ 0);
29              Next_Transition := !Transition_Num
30            else
31              assert (!Next_Transition ∈ ⟦Δ_d⟧)
32              assert (!Decremented == False)
33              ⋃_{(q,−ē_i,q')∈Δ_d} :
34                if y == ⟦(q, −ē_i, q')⟧ then
35                  assert (State == ⟦q⟧); assert (!Counter_Num == i);
36                  State := ⟦q'⟧; Decremented := True; Next_Transition := 0
37                else skip
38              assert* (!State ≠ ⟦q_f⟧)
```

Figure 8.3: The term encoding VASS coverability using only unit inputs

Figure 8.4: Prearena shape for $\vdash \beta \to \beta \to \beta \to \beta$

correspond to the different counters, with increments and decrements for a particular counter all being hereditarily justified by the same $\overset{\frown}{q_1 \; a_1}$-moves. A reset transition can then be encoded by changing a variable local to a $\overset{\frown}{q_1 \; a_1}$-thread in such a way that increments justified by it can no longer be decremented, and creating a new $\overset{\frown}{q_1 \; a_1}$-thread for subsequent increments for that counter. If we think of the extra layer as providing "bags" for the increments and decrements for each of the the k counters to go in, an example play is shown below.



Given an RVASS $(Q, n, q_0, \Delta_i \cup \Delta_d \cup \Delta_0)$, a $\mathsf{unit} \to \mathsf{int} \to \mathsf{int} \to \mathsf{unit}$ term encoding this RVASS is shown in Fig. 8.5, though we note the same trick as in Fig. 8.3 would allow us to adjust the term to any time $\beta \to \beta \to \beta \to \beta$. We describe how the various variables in the term are used, beginning with the global variables:

- *State*. This variable keeps track of the current state of the RVASS, as in previous encodings.

- *Setup_Done*. This variable keeps track of whether the initial creation of the $\overset{\frown}{q_1 \; a_1}$-threads for each counter has been completed.

- *Awaiting_New_Counter*. This keeps track of whether a new $\overset{\frown}{q_1 \; a_1}$-thread is needed, and if so which one. In particular, after a reset-transition fires (and

adjusts the internal state of a $\overset{\curvearrowleft}{q_1 \; a_1}$-thread), this will be set to the number of the counter which has been reset so that the next move creates a fresh $\overset{\curvearrowleft}{q_1 \; a_1}$-thread with appropriate internal state.

At the $\overset{\curvearrowleft}{q_1 \; a_1}$-level, we have the following variables:

- *Counter_Num*. Similar to previous encodings, this keeps track of which counter the increments justified by this thread are for.
- *Active*. This keeps track of whether this $\overset{\curvearrowleft}{q_1 \; a_1}$-thread has been 'reset'. When a reset transition for a counter occurs, if this is the current active $\overset{\curvearrowleft}{q_1 \; a_1}$-thread for that counter, this variable will be set to false. Once set to false, this prevents increments justified by this $\overset{\curvearrowleft}{q_1 \; a_1}$-thread from being created or decremented.

At the $\overset{\curvearrowleft}{q_2 \; a_2}$-level, we have only the *Decremented* variable, which operates as in the VASS-case.

Hence we get:

Theorem 8.1.3. *Observational equivalence of terms of type* $\vdash \beta \to \beta \to \beta \to \beta$ *is not primitive-recursive.*

8.1.3 VASS reachability to $\vdash \beta \to (\beta \to \beta) \to \beta$

We now show how the coverability reductions seen so far can be adjusted to give reductions to reachability if the final argument of the term is made a first-order argument. Arenas for this type are shown in Fig. 8.6. In particular the plays now take on a more complex structure than just sequences of $\overset{\curvearrowleft}{q_i \; a_i}$-pairs, and combine elements of $\mathrm{RML}_{\mathrm{CMA}}$-structure with the stack-like behaviour in $\mathrm{RML}_{\mathrm{O\text{-}Str}}$. When a q_2-move is played, P has the option of playing q_*. If P does, O then has the option of starting new $\overset{\curvearrowleft}{q_1 \; a_1}$-threads, or making more q_2-moves, before eventually playing a_*. Then P can play another q_* or simply a_2. By this means there may be several nested q_*-moves open at once. In this case the most recently asked q_*-move is the first to be answered. As we saw in Section 6.3.1.2, this behaviour is similar to that of a stack. We will exploit this kind of stack discipline in the VASS arguments above to ensure that every increment has been decremented.

```
1   let
2     State = ref ⟦q_0⟧
3     Setup_Done = ref False
4     Awaiting_New_Counter = ref 1
5   in
6     λx^unit.
7       let
8         Counter_Num = ref 0
9         Active = ref True
10      in
11        assert (!Awaiting_New_Counter ≠ 0)
12        Counter_Num := !Awaiting_New_Counter;
13        if (!Setup_Done == False) then
14          Awaiting_New_Counter := !Awaiting_New_Counter + 1
15        else Awaiting_New_Counter := 0
16        if (!Awaiting_New_Counter == d+1) then
17          Awaiting_New_Counter := 0; Setup_Done := True
18        else skip
19        λy^int.
20          let
21            Decremented = ref False
22          in
23            assert (!Awaiting_New_Counter = 0);
24            assert (!Active == True);  assert (y ∈ ⟦Δ_i⟧ ∪ ⟦Δ_0⟧)
25            ⋃_{(q,0_i,q')∈Δ_0} :
26              if y == ⟦(q,0_i,q')⟧then
27                assert (!State == ⟦q⟧); assert (!Counter_Num == i);
28                State := ⟦q'⟧; Active := False;
29                Awaiting_New_Counter := i
30              else skip
31            ⋃_{(q,ē_i,q')∈Δ_i} :
32              if y == ⟦(q,ē_i,q')⟧ then
33                assert (!State == ⟦q⟧); assert (!Counter_Num == i);
34                State := ⟦q'⟧
35              else skip
36            assert* (!State ≠ ⟦q_f⟧)
37            λz^int.
38              assert (!Awaiting_New_Counter = 0); assert (!Active == True)
39              assert (y ∈ ⟦Δ_d⟧); assert (!Decremented == False)
40              ⋃_{(q,−ē_i,q')∈Δ_d} :
41                if z == ⟦(q,−ē_i,q')⟧ then
42                  assert (State == ⟦q⟧); assert (!Counter_Num == i);
43                      State := ⟦q'⟧; Decremented := True
44                else skip
45              assert* (!State ≠ ⟦q_f⟧)
```

Figure 8.5: The term encoding RVASS coverability

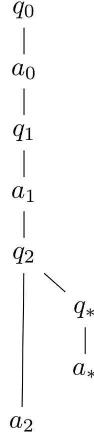


Figure 8.6: Prearena shape for $\vdash \beta \to (\beta \to \beta) \to \beta$

In particular, whilst a $\widehat{q_1 \; a_1}$-pair will still correspond to an increment of a counter, it will be immediately followed by $\widehat{q_2 \; q_*}$-moves justified by that $\widehat{q_1 \; a_1}$-thread. When the corresponding a_*-move is played, the term will check that the $\widehat{q_1 \; a_1}$-thread has had the *Decremented* flag set. Hence, for the play to complete all increments will have to be decremented. There is an additional complication to the encoding: where before we could ensure coverability by having one of the terms non-terminate if it ever reached the target state, we now need to check that we are not at the target state only at the very end of the run. To do this, we use another $\widehat{q_2 \; q_*}$-pair at the very beginning of the run: only at the corresponding a_*-move do we then check that the target state has not been reached.

The terms encoding VASS reachability are shown in Fig. 8.7. This is in fact at the type $\vdash \mathsf{int} \to (\mathsf{unit} \to \mathsf{int}) \to \mathsf{unit}$, though again the same trick as used in Fig. 8.3 could be applied to obtain terms of type $\vdash \mathsf{unit} \to (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$. We briefly discuss how the various variables are used in this term. We use global variables:

- *State*. As previously, this stores the current global states.

- *New_Threads_Allowed*. In the term we will get O to choose which transition is being applied at both the $\widehat{q_1 \; a_1}$-level and the $\widehat{q_2 \; a_2}$-level. However to get input from O at the $\widehat{q_2 \; a_2}$-level we must make the move q_*, which would allow O to start a new q_1 or q_2-thread out of turn. We thus use this global variable to keep track of whether such threads are permitted at this point in the run.

```
1  let
2    State = ref ⟦q_0⟧
3    New_Threads_Allowed = ref True
4    Expect_Inc = ref False
5    Run_Stage = ref 0
6  in
7    λx^int.
8      let
9        Counter_Num = ref 0
10       Incremented = ref False
11       Decremented = ref False
12     in
13       if (!Run_Stage == 0) then
14         Run_Stage := 1; Incremented := True; Decremented := True
15       else
16         assert (!Run_Stage == 2);  assert (!New_Threads_Allowed == True);
17         assert (!Expect_Inc == False);  assert (x ∈ ⟦Δ_i⟧);
18         ⋃_{(q,ē_i,q')∈Δ_i} :
19           if (x == ⟦(q, ē_i, q')⟧) then
20             assert (!State == ⟦q⟧); State := ⟦q'⟧;
21             Counter_Num := i; Expect_Inc := True
22           else skip
23         λf^{unit→int}.
24           if (!Run_Stage == 1) then
25             Run_Stage := 2;
26             f(skip);
27             Run_Stage := 3;
28             assert* (!State == ⟦q_f⟧)
29           else
30             assert (!New_Threads_Allowed == True);
31             assert (!Run_Stage == 2);
32             if (!Expecting_Inc == True) then
33               assert (!Incremented == False);
34               Incremented := True;  Expect_Inc := False;
35               f(skip);
36               Run_Stage := 3;
37               assert (!Decremented == True);
38             else
39               let
40                 y = ref 0
41               in
42                 New_Threads_Allowed := False;
43                 y := f(skip);
44                 New_Threads_Allowed := True;
45                 assert (y∈ ⟦Δ_d⟧); assert (!Decremented == False);
46                 ⋃_{(q,−ē_i,q')∈Δ_d} :
47                   if y == ⟦(q, −ē_i, q')⟧ then
48                     assert (State == ⟦q⟧); assert (!Counter_Num == i);
49                     State := ⟦q'⟧; Decremented := True
50                   else skip
```

Figure 8.7: The term encoding VASS reachability

- *Expect_Inc*. As mentioned above, an increment is now modelled by the moves $q_1\ a_1\ q_2\ q_*$. As O gets to make a choice after P plays a_1, we use this variable to keep track of whether we are expecting a q_2 - and if we are attempts by O to play q_1 will result in a failed run.

- *Run_Stage*. This keeps track of the setup and ending of the run. It starts at 0, and only increases as the run goes on. A value of 0 indicates we are expecting the initial $\widehat{q_1\ a_1}$-pair (this pair is merely some dummy moves, allowing us to play the subsequent $\widehat{q_2\ q_*}$-moves). A value of 1 means we are expecting the first $\widehat{q_2\ q_*}$-moves, which will be used to check whether the target state has been reached once all decrements have been made. A value of 2 will mean we are in the main part of the run, where O can generate new threads corresponding to the transitions being made. A value of 3 means the run is ending - the term is checking that all the increments have been decremented, and no new threads can be created.

We also have the following variables local to the $\widehat{q_1\ a_1}$-threads:

- *Counter_Num* and *Decremented*. These are used as in the coverability encodings already shown.

- *Incremented*. As mentioned, increments are now modelled by a $\widehat{q_1\ a_1}$-thread followed by moves $\widehat{q_2\ q_*}$. The *Expect_Inc* variable keeps track of whether we are expecting these $\widehat{q_2\ q_*}$-moves, but we also need to ensure that these moves are justified by the correct $\widehat{q_1\ a_1}$-thread. At most one $\widehat{q_1\ a_1}$-thread has the *Incremented* variable set to false at one time, and so we can always identify the most recent increment $\widehat{q_1\ a_1}$-thread.

Although this reduction does not provide us with a new – stronger – result, it demonstrates how the stack-like behaviour allows us to encode reachability problems, rather than simply coverability queries. In particular, it provides an illustration of one of the key ideas in the next result.

8.1.4 RVASS reachability to $\vdash \beta \to \beta \to (\beta \to \beta) \to \beta$

We have seen how $\vdash \beta \to \beta \to \beta \to \beta$ terms can encode RVASS coverability, and how $\vdash \beta \to (\beta \to \beta) \to \beta$ terms can encode VASS reachability. In fact we can combine these arguments to show that $\vdash \beta \to \beta \to (\beta \to \beta) \to \beta$ terms can encode RVASS reachability.

Theorem 8.1.4. *Given a RVASS $\mathcal{A} = (Q, k, \Delta_i \cup \Delta_d \cup \Delta_0, q_0)$ and target state $q_f \in Q$, there are* RML*-terms* $\vdash M, M' : \mathsf{unit} \to \mathsf{int} \to (\mathsf{unit} \to \mathsf{int}) \to \mathsf{unit}$ *such that $M \cong M'$ iff there is a run of $\mathcal{A}$ reaching configuration $(q_f, \bar{0})$.*

Proof. This is a simple combination of the ideas from the RVASS coverability and VASS reachability arguments above. We provide the term describing this in Fig. 8.8. □

Crucially, this gets us the following result:

Corollary 8.1.5. *The observational equivalence problem for* RML *terms of type* $\vdash \beta \to \beta \to (\beta \to \beta) \to \beta$ *is undecidable.*

8.2 Undecidability at left-hand side types

So far in this chapter we have been using types on the right-hand side of the typing judgement to obtain hardness and undecidability results. However, we have not investigated what types of free variables are sufficient to obtain undecidability.

In [40] Hopkins showed that if observational equivalence of terms at any type $\vdash \theta$ could be reduced to observational equivalence of terms of type $\theta \to \beta \vdash \beta$, and we can use the hardness and undecidability arguments presented in the first half of this chapter with this argument to obtain the hardness results shown in Table 8.1.

However, several open questions still remain. In particular, whilst in Chapter 7 we were able to obtain decidability results for type fragments involving some second-order free variables, it was unclear whether it might be possible to extend these results to permit all second-order types for free variables. In fact, free variables at some second-order types are sufficient to obtain undecidability. We spend the rest of this chapter proving the following theorem.

```
1   let
2     State = ref ⟦q0⟧; New_Threads_Allowed = ref True; Expect_Inc = ref False;
3     Run_Stage = ref 0;  Awaiting_New_Counter = ref 1
4   in
5     λx^unit.
6       let
7         Counter_Num = ref 0;  Active = ref True
8       in
9         assert (!Awaiting_New_Counter ≠ 0); assert (!New_Threads_Allowed == True);
10        if (!Run_Stage == 0) then
11          Active := False; !Run_Stage == 1
12        else if (!Run_Stage == 3) then
13          Counter_Num := !Awaiting_New_Counter; Awaiting_New_Counter++
14          if (!Awaiting_New_Counter == d+1) then
15            Awaiting_New_Counter := 0; Run_Stage := 4
16          else skip
17        else
18          assert (!Run_Stage == 4);
19          Counter_Num := !Awaiting_New_Counter; Awaiting_New_Counter := 0
20        λy^int.
21          let
22            Incremented = ref False;  Decremented = ref False
23          in
24            if (!Run_Stage == 1) then
25              Run_Stage := 2; Incremented := True; Decremented := True
26            else
27              assert (!Run_Stage == 4); assert (!Active == True); assert (!Expect_Inc == False);
28              assert (!New_Threads_Allowed == True);  assert (y ∈ ⟦Δi⟧ ∪ ⟦Δ0⟧);
29              ⋃_{(q,ē_i,q')∈Δ_i} :
30                if (y == ⟦(q, ē_i, q')⟧) then
31                  assert (!State == ⟦q⟧); State := ⟦q'⟧;
32                  Counter_Num := i; Expect_Inc := True
33                else skip
34              ⋃_{(q,0̄_i,q')∈Δ_i} :
35                if (y == \sem{(q, \bar{0_i}, q')}) then
36                  assert (!State == ⟦q⟧); assert (!Counter_Num == i);
37                  State := \sem{q'};  Active := False;  Awaiting_New_Counter := i
38              λf^unit→int.
39                if (!Run_Stage == 2) then
40                  Run_Stage := 3;  f(skip);
41                  Run_Stage := 5;  assert* (!State == ⟦q_f⟧)
42                else
43                  assert (!New_Threads_Allowed == True);  assert (!Run_Stage == 4);
44                  assert (!Active == True);
45                  if (!Expecting_Inc == True) then
46                    assert (!Incremented == False); Incremented := True;
47                    f(skip);  Run_Stage := 5;
48                    assert (!Decremented == True OR !Active == False);
49                  else
50                    let
51                      z = ref 0
52                    in
53                      New_Threads_Allowed := False;
54                      z := f(skip);
55                      New_Threads_Allowed := True;
56                      assert (z ∈ ⟦Δd⟧); assert (!Decremented == False)
57                      ⋃_{(q,−ē_i,q')∈Δ_d} :
58                        if z == ⟦(q, −ē_i, q')⟧ then
59                          assert (State == ⟦q⟧); assert (!Counter_Num == i);
60                          State := ⟦q'⟧; Decremented := True
61                        else skip
```

Figure 8.8: The term encoding RVASS reachability

Table 8.1: LHS hardness results obtainable from Section 8.1.

Representative Type Sequent	Complexity
$(\beta \to \beta \to \beta) \to \beta \vdash \beta$	ExpSpace-hard
$(\beta \to \beta \to \beta \to \beta) \to \beta \vdash \beta$	$\notin$ PR
$(\beta \to (\beta \to \beta) \to \beta) \to \beta \vdash \beta$	VASS reachability-hard
$(\beta \to \beta \to (\beta \to \beta) \to \beta) \to \beta \vdash \beta$	Undecidable

Theorem 8.2.1. *Observational equivalence of* RML *terms of type*

$$(\beta \to \beta \to \beta \to \beta) \to \beta \to \beta \to \beta \vdash \beta$$

is undecidable.

8.2.1 Queue Machines

We will prove this undecidability result via reduction to queue machines. These are deterministic finite-state devices augmented with a queue over a finite alphabet. The states are partitioned into enqueuing states and dequeuing states. If in an enqueuing state the next enqueued letter and next state are determined just by the current state of the device. If in an dequeuing state the next state is determined by the next letter in the queue (which is removed) and the current state. The machine halts just if it ever reaches an empty queue.

Definition 8.2.1. A *queue machine* is a tuple $\langle Q, Q_E, Q_D, q_0, \Sigma, \sigma_0, \delta_E, \delta_D \rangle$ where Q is a finite set of states with $Q = Q_E \uplus Q_D$, $q_0 \in Q$ is the initial state, Σ is a finite alphabet, $\sigma_0 \in \Sigma$ is the starting queue element, and $\delta_E : Q_E \to Q \times \Sigma$ and $\delta_D : Q_D \times A \to Q$ are the enqueuing and dequeuing functions respectively.

A configuration of a queue machine is a pair $(q, S) \in Q \times \Sigma^*$. The initial configuration is (q_0, σ_0). If in configuration (q, S) where $q \in Q_E$, the machine transitions to $(\pi_1(\delta_E(q)), S \cdot \pi_2(\delta_E(q)))$ (where π_1 and π_2 are the standard projection functions). If in configuration $(q, \sigma \cdot S)$ where $q \in Q_D$ and $\sigma \in \Sigma$ the machine transitions to configuration $(\delta_D(q, \sigma), S)$. The machine is said to *halt* just if it ever reaches a state (q, ϵ) for any $q \in Q$.

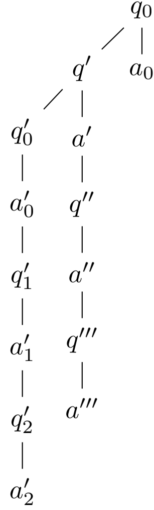


Figure 8.9: Prearena shape for $(\beta \to \beta \to \beta \to \beta) \to \beta \to \beta \to \beta \vdash \beta$

Queue machines are well known to be Turing-complete, see e.g. [51]. This argument is easy to see, and proceeds by storing the tape of the Turing machine in the queue with a special marker for the head. Changes can by made to the tape by "rolling over" the tape, dequeuing and enqueuing each element to be left unchanged. Hence, the halting problem for queue machines is undecidable.

8.2.2 Undecidability at $(\beta \to \beta \to \beta \to \beta) \to \beta \to \beta \to \beta \vdash \beta$

Here we show how the halting problem for queue machines may be encoded in observational equivalence just using an order 2 free variable. The type and corresponding prearena is shown in Fig. 8.9.

8.2.2.1 Shape of the Encoding

The key idea for this encoding is how the queue is encoded: elements of the queue will be represented by $\overset{\frown}{q'_0 \;\; a'_0}$-threads, with suitable links between them to preserve the order of the queue. These links are constructed to make use of the visibility conditions on the play.

In fact, we will use other $\overset{\frown}{q'_0 \;\; a'_0}$-threads to build the links between the $\overset{\frown}{q'_0 \;\; a'_0}$-threads representing elements in the queue. Thus we will use two different types $\overset{\frown}{q'_0 \;\; a'_0}$-threads, and for clarity we refer to these as M-$\overset{\frown}{q'_0 \;\; a'_0}$-threads and N-$\overset{\frown}{q'_0 \;\; a'_0}$-threads. M-$\overset{\frown}{q'_0 \;\; a'_0}$-threads will be the ones to hold information about the elements of

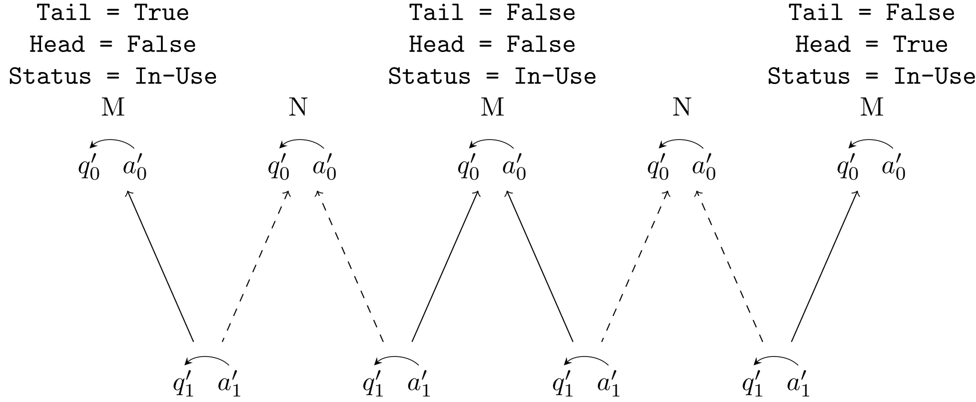
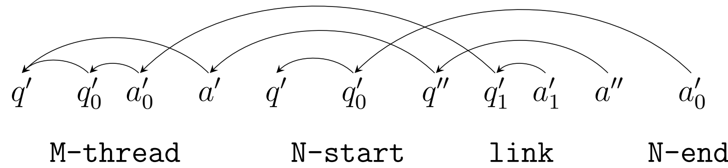


Figure 8.10: Links storing counter values. Solid lines indicate justification pointers, dashed lines indicate containment.

the queue, including the Σ-value of that element, and whether it is currently the Head or the Tail of the queue. This information is stored in local variables of that thread. N-$\overleftarrow{q'_0 \; a'_0}$-threads will be used to perform updates to the queue, simulating transitions of the queue-machine, and maintain the order of the queue elements. Crucially M-$\overleftarrow{q'_0 \; a'_0}$-threads will be linked to N-$\overleftarrow{q'_0 \; a'_0}$-threads by $\overleftarrow{q'_1 \; a'_1}$-moves. A $\overleftarrow{q'_1 \; a'_1}$-play can link one M-$\overleftarrow{q'_0 \; a'_0}$-thread with one N-$\overleftarrow{q'_0 \; a'_0}$-thread by being justified by the M-$\overleftarrow{q'_0 \; a'_0}$-thread, and occuring "within" the occurrence of the N-$\overleftarrow{q'_0 \; a'_0}$-thread. We illustrate what is meant by this in the following subplay:



This encoding of a queue is depicted in Fig. 8.10.

We now describe how the queue is setup and how elements are added and removed from the queue. In doing so we will refer to the local variables of the M-threads. These are:

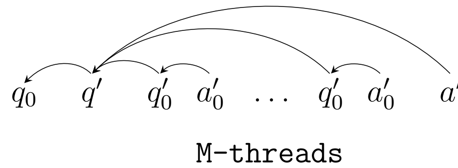
- `Head`. This is a Boolean variable describing whether the queue element represented by this M-thread is currently at the head of the queue.

- `Tail`. This is a Boolean variable describing whether the queue element represented by this M-thread is currently at the tail of the queue.

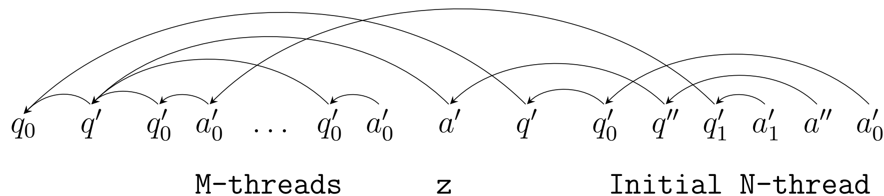
- `Sigma_Value`. This variable stores which Σ-value the queue element represented by this M-thread is.
- `Status`. This variable is used to keep track of whether this M-thread is currently being used to represent an element of the queue, has already been used to do so, or has not yet been used to do so. These states are represented by values *In-Use*, *Expired*, and *Unused* respectively.

Note that in the below O will often have a choice of justifier, or move, but we make sure that O makes the choices we desire by use of assertions on local variables. If the assertion fails the play will not complete, and so P's strategy restricts itself to those instances where O makes the correct choices.

Initial setup. The play starts with O playing the move q_0 and P replying with the move q'. We then require O to generate a number of $\overset{\curvearrowleft}{q'_0 \;\; a'_0}$-threads, to be used later in the play. These are the M-threads which will be linked, later in the play, to encode the queue. O will non-deterministically decide when to stop generating these threads, and play a'. At this point the play will look like this:



Next, P will play another q'-move and the queue will be set up with the initial element σ. This will consist of O playing a q'_0 move, and then this $\overset{\curvearrowleft}{q'_0 \;\; a'_0}$-thread being linked to one of the already created M-threads as previously described. This link will also change the M-thread's internal variables to have both the `Head` and `Tail` variables set to True, and to store in an internal variable which σ-value this queue element is. Hence at this stage the play will be roughly as follows (although it needn't be the first M-thread that justifies the $\overset{\curvearrowleft}{q'_1 \;\; a'_1}$-thread):



Following this further N-$\overset{\curvearrowleft}{q'_0 \ a'_0}$ and $\overset{\curvearrowleft}{q'_1 \ a'_1}$-moves are played, enacting the transitions of the machine. Note that we have labelled the first a'-move above as `z`, as we will need to refer to this move below.

Enqueuing elements. To perform a transition that enqueues an element is reasonably straightforward. In a new N-$\overset{\curvearrowleft}{q'_0 \ a'_0}$-thread, P plays a q''-move (justified by the move `z`), which we will term this N-thread's q''_{prev}-move, after which O plays a $\overset{\curvearrowleft}{q'_1 \ a'_1}$-thread justified by a M-$\overset{\curvearrowleft}{q'_0 \ a'_0}$-thread. In this $\overset{\curvearrowleft}{q'_1 \ a'_1}$-thread the term checks that the parent M-$\overset{\curvearrowleft}{q'_0 \ a'_0}$-thread has local state `Status`=In-Use, `Head`=True, and changes `Head` to False. After the $\overset{\curvearrowleft}{q'_1 \ a'_1}$-thread, O plays a'', and P plays another q''-move (again justified by `z`), which we will call this N-thread's q''_{next}-move, and again O starts a $\overset{\curvearrowleft}{q'_1 \ a'_1}$-thread justified by a M-$\overset{\curvearrowleft}{q'_0 \ a'_0}$-thread. This time P checks that this M-thread hasn't already been used (since the play of `z`) by asserting the local variable `Status`=Unused and then changes `Status` to In-Use and sets its local `Head` variable to True.

Dequeuing elements. To dequeue an element O begins by playing a move q'_1, justified by a N-$\overset{\curvearrowleft}{q'_0 \ a'_0}$-thread. Crucially, we assume (and will enforce) that this N-$\overset{\curvearrowleft}{q'_0 \ a'_0}$-thread links the current tail of the queue to the next element of the queue. First, P will check whether the current tail is not also the head of the queue. If this is the case then the dequeue action would result in an empty queue so the machine would halt. Otherwise, the dequeue action can proceed as normal. In this case P plays a q'''-move, justified by the q''_{prev}-a''-pair of the parent N-thread. O now plays a q'_2 justified by an M-$\overset{\curvearrowleft}{q'_1 \ a'_1}$-thread. P checks that this has local variables `Status`=In-Use and `Tail`=True, and changes these to Expired and False respectively. Then these q'_2 and q''' moves have their answer moves played. Then P plays another q'''-move, this time justified by the parent N-thread's q''_{next}-a''-pair, and O plays another q'_2-move. P then changes this M-thread's local `Tail` variable from False to True.

Crucially in the above, the choice of the justifying moves for the q'''-moves restricted O's visibility, and gave O no choice in which M-thread was used to justify the subsequent q'_2-move. In this way the order of the queue elements is maintained.

Checking for empty queue. As mentioned above, before dequeuing we check that the queue has more than one element in it. We do this by checking whether the current M-thread with `Head`=True also has `Tail`=True. This is fairly straightforward, as P plays a q''-move justified by `z`, and O then plays a q'_1-move. We then assert that this has local variables `Status`=In-Use and `Tail`=True and the outcome depends on the value of `Head`.

8.2.2.2 The Term Encoding a Queue Machine

We now move to actually presenting the term that encodes a queue machine. This term has the property that it is observationally equivalent to Ω iff the encoded queue machine doesn't halt. We give the term for a queue machine $\langle\, Q, Q_E, Q_D, q_0, \sigma_0, \delta_E, \delta_D\,\rangle$ in Figs. 8.11 to 8.13. In particular note that we have separate terms corresponding to P's play in the M-threads and N-threads.

We begin by describing the overall structure of the term. This is shown in Fig. 8.11. In broad terms, after the global variables are set up, we have the construction $\mathbf{let}\, x = vM\,\mathbf{in}\,\ldots$ (where v is the unique free variable). This corresponds to P playing the move q', and then playing as M until O plays the a'-move. Crucially, P will resume playing as M if (later in the term) x is (partially) evaluated. After this first a'-move, we have a similar construction $\mathbf{let}\, y = vN\,\mathbf{in}\,\ldots$. This corresponds to P playing another q'-move and then playing as N until O plays a'. It is this part of the play in which the simulation of the queue machine will take place, and communication between the M and N-threads will take place by P partially or fully evaluating x as part of playing as N.

We summarise the global variables and the roles they play in the term.

- `State`. This variable is used to store which state of the simulated queue machine is in. As well as taking values of states, we also have two special values it may take: *init*, to signal the beginning of the run, and *EmptyQueue*, to signal that the queue has emptied and so the machine halts. To allow us to encode states (and these special values) in the int-type, we fix an injective function $[\![-]\!] : Q \cup \{init, EmptyQueue\} \to \mathsf{int}$.

- `Current_Sigma`. This variable is used as a temporary store when enqueuing or dequeuing to remember the Σ-value that is being enqueued or has been

```
1   let
2     State = ref ⟦init⟧
3     Current_Sigma = ref −
4     Control = ref ⟦NewThreads⟧
5   in
6     let
7       x^{unit→unit→unit} = v M
8     in
9       let
10        y^{unit→unit→unit} = v N
11      in
12        assert (!State == ⟦EmptyQueue⟧)
```

Figure 8.11: Shape of the term encoding queue machines

dequeued. It also takes a special value $-$, when the variable is not in use. Hence, as this variable is of type int, we use an injective function $[\![-]\!] : \Sigma \cup \{-\} \to \mathsf{int}$.

- `Control`. This variable is used to play in N to control the behaviour of play in M, when N partially or fully evaluates the variable x. It can take a several values (again stored in type intusing an injective function), which we list here:
  - *NewThreads*. This value is taken at the start, and permits O to start any number of $\overset{\curvearrowleft}{q'_0\ a'_0}$-threads. Note that it is never again set to this value, so once changed O cannot create any further M-$\overset{\curvearrowleft}{q'_0\ a'_0}$-threads.
  - *Skip*. This value is used when setting up the queue to prevent O from playing in M-threads.
  - *RemoveHead*, *NewHead*, *NewHeadTail*. These values are used as instructions from N to M, to update the queue elements accordingly (using $\overset{\curvearrowleft}{q'_1\ a'_1}$-moves).
  - *Done*. This value is used to signify that M has completed the instructed action. This both prevents multiple M-actions from being taken when control passes to M, and ensures that at least one such action is taken.
  - *CheckHeadTail*, *Head = Tail*, *Head ≠ Tail*. *CheckHeadTail* is used when N requires M to check whether the head of the queue is also the tail. The two other values here are used to convey M's response back to N.
  - *RemoveTail*, *UpdateTail*. These values are used as instructions from N to M, to update the queue elements accordingly (using $\overset{\curvearrowleft}{q'_2\ a'_2}$-moves).

The term for M is shown in Fig. 8.12. This term is a reasonably straightforward implementation of the queue-update behaviour described in Section 8.2.2.1, using the `Control` variable described above.

The term for N is given in Fig. 8.13. Note that if O ever provides input for the final argument of N (corresponding to playing q_2' justified by an N-thread), this immediately gives non-termination. Plays in N will therefore only consist of $\overset{\curvearrowleft}{q_0' \quad a_0'}$ and $\overset{\curvearrowleft}{q_1' \quad a_1'}$-moves. $\overset{\curvearrowleft}{q_0' \quad a_0'}$-moves (corresponding to providing input for the first argument) will be used to model enqueuing transitions (and the initial queue setup), whilst $\overset{\curvearrowleft}{q_1' \quad a_1'}$-moves (providing input for the second argument) will model dequeuing transitions. We describe how each of these simulations work.

When enqueuing matters are slightly complicated by the fact that the same moves are used to setup the queue, and so we have several conditional clauses. First, we check that the current state is an enqueuing state (or we are setting up the queue for the first time). Next we will change the current queue head's `Head` variable from True to False. This is not necessary if the queue is being setup, and so we conditionally set `Control` to *Skip* or *RemoveHead* as appropriate. We partially evaluate x, allowing control to pass to M, and check that the appropriate M-action (if any) has been taken by checking the variable `Control`$=[\![Done]\!]$. Then we move onto the creating the new head of the queue. For this we put the appropriate Σ-value in the global variable `Current_Sigma` and (depending on whether this is the new Head and Tail or merely new Head) set the variable `Control` appropriately. We then partially evaluate x, again passing control to M, to allow the necessary updates to be made. Finally we update the current state of the machine, in the global variable `State`.

When performing a dequeuing transition, we first check the current state is a dequeuing state, and then check whether this dequeue would take us to an empty queue - this is done by passing control to M by partially evaluating x. If it would, we simply set the global variable `State` to $[\![EmptyQueue]\!]$, signalling that the machine halts. Otherwise, we continue and first partially evaluate z_{prev} to remove that element from the queue. In doing so M will update the global variable `Current_Sigma` to give the Σ-value of that queue element. Then by partially evaluating z_{next} we let M update that queue element to have `Tail`=True. Note in the above partial evaluations the behaviour of M will guarantee that the N-$\overset{\curvearrowleft}{q_0' \quad a_0'}$-thread justifying the current

```
1  λx1^unit.
2    let
3      Head = ref False
4      Tail = ref False
5      Status = ref ⟦Unused⟧
6      Sigma_Value = ref ⟦−⟧
7    in
8      assert (!Control == ⟦NewThreads⟧)
9      λx2^unit.
10       if (!Control == ⟦RemoveHead⟧) then
11         assert (!Head == True); assert (!Status == ⟦In − Use⟧);
12         Head := False;
13         Control := ⟦Done⟧
14       else if (!Control == ⟦NewHead⟧) then
15         assert (!Status == Unused); Status := ⟦In − Use⟧;
16         Head := True; Sigma_Value := !Current_Sigma;
17         Control := ⟦Done⟧
18       else if (!Control == ⟦NewHeadTail⟧) then
19         assert (!Status == Unused); Status := ⟦In − Use⟧;
20         Head := True; Sigma_Value := !Current_Sigma; Tail := True;
21         Control := ⟦Done⟧
22       else if (!Control == ⟦CheckHeadTail⟧) then
23         assert (!Status == In−Use); assert (!Tail == True);
24         if (!Head == True) then
25           Control := ⟦Head = Tail⟧
26         else
27           Control := ⟦Head ≠ Tail⟧
28       else Ω
29       λx3^unit.
30         if (!Control == ⟦RemoveTail⟧) then
31           assert (!Tail == True); assert (!Status == ⟦In − Use⟧);
32           Tail := False; Status := ⟦Expired⟧;
33           Current_Sigma := !Sigma_Value;
34         else if (!Control == ⟦UpdateTail⟧) then
35           assert (!Tail == False); assert (!Status == ⟦In − Use⟧);
36           Tail := True;
37         else Ω
```

Figure 8.12: The term M in the term encoding queue machines

```
1   λy₁^unit.
2     assert (!State ∈ {⟦init⟧} ∪ Q_E);
3     if (!State == ⟦init⟧) then
4       Control := ⟦Skip⟧
5     else
6       Control := ⟦RemoveHead⟧
7     let
8       z_prev = x(skip)
9     in
10      if (!State == ⟦init⟧) then
11        assert (!Control == ⟦Skip⟧);
12        Current_Sigma := ⟦σ_0⟧; Control := ⟦NewHeadTail⟧
13      else
14        assert (!Control == ⟦Done⟧);
15        Current_Sigma := ⟦π_1(δ_E(!State))⟧; Control := ⟦NewHead⟧
16      let
17        z_next = x(skip)
18      in
19        assert (!Control == ⟦Done⟧);
20        Current_Sigma := −;
21        if (!State == ⟦init⟧) then
22          State := ⟦q_0⟧
23        else
24          State:= ⟦π_1(δ_E(!State))⟧
25
26        λy₂^unit.
27          assert (!State ∈ Q_D)
28          Control := ⟦CheckHeadTail⟧
29          x(skip);
30          if (!Control == ⟦Head = Tail⟧ then
31            State := ⟦EmptyQueue⟧
32          else
33            assert (!Control == ⟦Head ≠ Tail⟧;
34            Control := ⟦RemoveTail⟧;
35            z_prev(skip); assert (!Control == ⟦Done⟧);
36            Control := ⟦UpdateTail⟧;
37            z_next(skip); assert (!Control == ⟦Done⟧);
38            State := ⟦δ_D(!State, !Current_Sigma)⟧;
39            Current_Sigma := −;
40
41            λy₃^unit.
42              Ω
```

Figure 8.13: The term N in the term encoding queue machines

N-$\overset{\frown}{q_1' \quad a_1'}$-thread being played is the N-thread linking the previous and current tail elements of the queue (if O had not chosen the correct N-thread, the result would be non-termination in M). Finally, we can update the `State` variable, using the current state and the dequeued sigma value stored in `Current_Sigma`.

8.3 Summary

We began this chapter with three key questions: (i) is observational equivalence decidable for all of $\mathrm{RML}_{2\vdash 1}$? (ii) can the fragments of $\mathrm{RML}_{2\vdash 1}$ for which observational equivalence is decidable, be decided more efficiently than we did in Chapter 7? (iii) is observational equivalence of terms of types $\vdash \beta \to \cdots \to \beta \to (\beta \to \beta) \to \beta$ decidable?

In Section 8.2 we showed that the answer to question (i) is no, as free variables of type $(\beta \to \beta \to \beta \to \beta) \to \beta \to \beta \to \beta$ are sufficient to encode Turing-complete queue machines.

In Section 8.1 we first showed that observational equivalence at type $\vdash \beta \to \beta \to \beta$ is ExpSpace-hard, by reduction from VASS-coverability. We then showed that for more complex first-order types (i.e. arity 3 or greater), observational equivalence is non-primitive recursive, and so we provide a partial answer to question (ii).

We then showed that the simplest type which question (iii) asked about – $\vdash \beta \to (\beta \to \beta) \to \beta$ – is at least as hard as VASS-reachability. We extended this argument to show that lengthening these types to have arity greater than 2 yields undecidability, as reset VASS reachability then becomes encodable. This provides most of an answer to question (iii): the only remaining part of the question is whether observational equivalence of terms of type $\vdash \beta \to (\beta \to \beta) \to \beta$ is decidable or not. We address this in the next chapter.

Chapter 9

$\mathrm{RML}_{\mathrm{VPCMA}}$

In Chapter 7 we saw that observational equivalence is decidable for all first-order RML terms without free variables. Previous work in [41, 40] showed, by reduction to equivalence of visibly pushdown automata, that all second-order RML terms with only one argument and no free variables also have a decidable observational equivalence problem. They also showed that at third order, or second-order terms with a first-order argument that is not the final argument, observational equivalence becomes undecidable. In the previous chapter we showed that second-order terms with more than two arguments also have an undecidable observational equivalence problem. Hence, the only remaining types without free variables for which the decidability of observational equivalence is unknown are of the form $\vdash \beta \to (\beta \to \cdots \to \beta) \to \beta$, and these we saw to have an observational equivalence problem at least as hard as VASS reachability.

In this chapter we will show that observational equivalence at these types is decidable if and only if the emptiness problem for visibly pushdown class memory automata (VPCMA) is also decidable. Recall that in Chapter 5 we introduced VPCMA, and showed their emptiness problem to be equivalent to extended branching VASS [45].

In fact, we will define a fragment which we call $\mathrm{RML}_{\mathrm{VPCMA}}$, which permits some free variables in terms of type $\beta \to (\beta \to \cdots \to \beta) \to \beta$, and show for this fragment observational equivalence is equivalent to emptiness of SVPCMA. We will thus spend this chapter proving the following result:

Theorem 9.0.1. *Observational equivalence of* $\mathrm{RML}_{\mathrm{VPCMA}}$ *is equivalent to the emptiness problem for SVPCMA.*

In showing observational equivalence can be reduced to emptiness of SVPCMA we will adopt a very similar approach to that of Chapter 7. Namely, we will give an algorithm that, given a term M in canonical form and within $\mathrm{RML}_{\mathrm{VPCMA}}$, constructs a deterministic weak SVPCMA, $\mathcal{A}_M$, recognising, as a language, a (unique) representation of $[\![M]\!]$. Hence, observational equivalence of terms M and N is then reduced to checking whether $\mathcal{L}(\mathcal{A}_M) = \mathcal{L}(\mathcal{A}_N)$. Recall, from Chapter 5, that for deterministic weak SVPCMA, this problem is reducible to emptiness checking. This reduction will combine the ideas of using data values for dealing with terms of arity greater than one (which we saw in Chapter 7, and was first used in [40]), together with the ideas of using visibly pushdown automata to deal with first-order arguments, introduced in [41] in deciding the "O-strict" fragment of RML.

We begin this chapter with a full definition of $\mathrm{RML}_{\mathrm{VPCMA}}$, and discussion of the shape of plays of this fragment. We then give a reduction, in the style of those in Chapter 7, from $\mathrm{RML}_{\mathrm{VPCMA}}$ to SVPCMA. Finally, we show the reverse reduction, using several of the ideas we saw in Section 8.1.

9.1 $\mathrm{RML}_{\mathrm{VPCMA}}$ fragment definition

As previously mentioned, the only RML terms without free variables for which we don't have a firm decidability or undecidability result for the observational equivalence problem are those terms with type of the form $\beta \to (\beta \to \cdots \to \beta) \to \beta$. That is, the type is an O-strict type with an additional ground type argument as the first argument. These types will be the ones permitted on the right-hand side of the turnstile in $\mathrm{RML}_{\mathrm{VPCMA}}$.

For the left-hand side of the typing judgement, i.e. the types of the free variables, we will permit the same types as in the O-strict fragment. That is, all types for which the O-pointers are determined by the underlying sequence of moves. This means that the parts of plays corresponding to interrogating the free variables will not have to have their O-pointers represented by the automaton.

Definition 9.1.1. We define the fragment $\mathrm{RML}_{\mathrm{VPCMA}}$ as being typed terms in context of the form

$$x_1 : \Theta_3, \ldots, x_n : \Theta_3 \vdash M : \Theta_0 \to \Theta_2$$

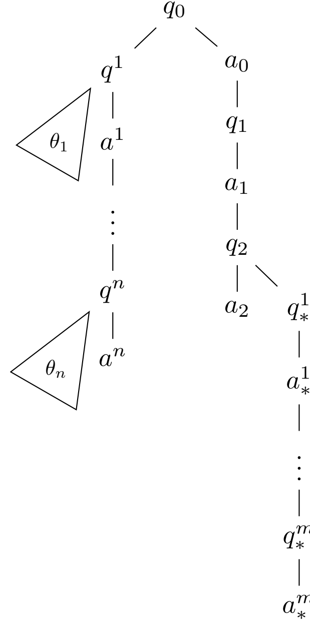


Figure 9.1: Shape of prearena for $\Theta_1 \to \cdots \to \Theta_n \to \beta \vdash \beta \to (\beta \to \cdots \to \beta) \to \beta$

where

$$\Theta_0 ::= \mathsf{unit} \mid \mathsf{int} \qquad\qquad \Theta_1 ::= \Theta_0 \mid \Theta_0 \to \Theta_1 \mid \mathsf{int\ ref}$$

$$\Theta_2 ::= \Theta_0 \mid \Theta_1 \to \Theta_0 \mid \mathsf{int\ ref} \qquad\qquad \Theta_3 ::= \Theta_0 \mid \Theta_2 \to \Theta_3 \mid \mathsf{int\ ref}$$

The shape of the prearenas for terms in this fragment is shown in Fig. 9.1.

Plays in these prearenas have a pushdown flavour to them in the same way as in $\mathrm{RML}_{\text{O-Str}}$: playing a q_*^i-move gives O an opportunity to begin new threads and these must be completed following the stack discipline. However, plays in these prearenas also have sub-threading, in the same way as we saw in Chapter 7. That is, each $\overset{\frown}{q_1\ a_1}$-pair can be thought of as a thread, and each q_2-move then behaves as a sub-thread to the justifying $\overset{\frown}{q_1\ a_1}$-moves. Further, 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 sub-thread instead. Hence, the plays for this fragment will consist of the initial $\overset{\frown}{q_0\ a_0}$-pair, followed by an interleaving of threads where each thread consists of a $\overset{\frown}{q_1\ a_1}$-pair, followed by some number of $\overset{\frown}{q_2\ a_2}$-subthreads. This interleaving may switch between threads after any a_1, a_2, or q_*^i-move, and where a q_*^i-move is made that q_2-subthread is only returned to subject to the stack discipline.

However, the visibility condition on plays gives rise to an additional restriction that we have not seen in either $\mathrm{RML}_{\text{O-Str}}$ or $\mathrm{RML}_{2\vdash 1}$: if a $\overset{\frown}{q_1\ a_1}$-pair is played after

a q_*^i-move and before that move has been popped, then once the pop-move is played, the "contained" $\overset{\curvearrowleft}{q_1 \; a_1}$-pair will no longer be visible when O gets to choose whether to create a new thread or sub-thread, and hence cannot justify further q_2-moves. This restriction, which we cannot capture simply using data values by themselves, corresponds precisely to the *scoping* condition of SVPCMA, and hence we will construct SVPCMA representing $\mathrm{RML}_{\mathrm{VPCMA}}$-terms.

9.2 Reduction to SVPCMA

The key ideas of this are a merging of the ideas in the $\mathrm{RML}_{\text{O-Str}}$ and $\mathrm{RML}_{2\vdash 1}$ cases. As ever the moves form the basis of the finite alphabet. As in the VPA case, we use the q_*^i moves as the push-moves, and the a_*^i moves as pops. As in the CMA case, each q_1-move gets its own data value corresponding to the name of that 'thread' - each move hereditarily justified by a q_1-move shares its data value.

9.2.1 Encoding pointers

As already mentioned, encoding the strategies for $\mathrm{RML}_{\mathrm{VPCMA}}$ as a language will require encoding both some P-pointers and some O-pointers. In both cases, it is only the pointers of question moves that require encoding, since answer moves' pointers are determined by well-bracketing.

The P-pointers will be encoded using the marking technique used in reducing $\mathrm{RML}_{\text{O-Str}}$ to VPA [41, 40], which we already saw Section 6.3.1.2, and used again in Section 7.3. The O-pointers will be encoded using data values. This is similar to the use of data values in Chapter 7, though in this case we do not have multiple layers of data values. In particular, data values will be used as follows:

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

9.2.2 Constructions

The key idea of this construction is that strategies of terms in $\mathrm{RML}_{\mathrm{VPCMA}}$ are, after the unique a_0-move, essentially interleavings of strategies of terms in $\mathrm{RML}_{\textsf{O-Str}}$ (with some additional restrictions based on the top level structure of the term). Since we can recognise $\mathrm{RML}_{\textsf{O-Str}}$-terms with VPA, 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.

In the rest of this section we briefly describe the inductive construction. As before, we induct over the canonical forms, and each term $\Gamma \vdash M$ will be represented by a family of weak deterministic SVPCMA $(\mathcal{A}_i^M)$, indexed by the initial move i.

The simple constructions: $()$, i, x^β, $\mathbf{succ}(x^\beta)$, $\mathbf{pred}(x^\beta)$, $\mathbf{if}\, x^\beta\, \mathbf{then}\, \mathbb{C}\, \mathbf{else}\, \mathbb{C}$, $x^{\textsf{int ref}} := y^{\textsf{int}}$, $!x^{\textsf{int ref}}$, and $\mathbf{mkvar}(\lambda x^{\textsf{unit}}.\mathbb{C}, \lambda y^{\textsf{int}}.\mathbb{C})$, $\mathbf{while}\, \mathbb{C}\, \mathbf{do}\, \mathbb{C}$ are all covered by the $\mathrm{RML}_{\textsf{O-Str}}$ constructions. Similarly in the below we assume the term being constructed is of type $\beta \to (\beta \to \cdots \to \beta) \to \beta$, as otherwise it is covered by the $\mathrm{RML}_{\textsf{O-Str}}$ work. The $\mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}\, M$ case is another construction of restriction to good-variable behaviour and then hiding of the moves corresponding to the variable. This is straightforwardly similar to the construction seen in Section 7.3. The $\mathbf{let}\, x = zy^\beta\, \mathbf{in}\, M$, $\mathbf{let}\, x = z\; \mathbf{mkvar}(\lambda y_1^{\textsf{unit}}.N_1, \lambda y_2^{\textsf{int}}.N_2)\, \mathbf{in}\, M$, and $\mathbf{let}\, x = z\, (\lambda y.N)\, \mathbf{in}\, M$ cases are adaptations of the corresponding cases in the O-strict constructions from [40]. 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 the two remaining cases in more detail:

9.2.2.1 $\lambda x.M$

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 $\beta \in \{\textsf{int}, \textsf{unit}\}$.

Plays in $[\![\lambda x.M]\!]$ are as follows: 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, $(\mathcal{A}_i^M)$, recognising the strategies of $[\![M]\!]$. 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 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 $q_1 \xrightarrow{a} q_2$ inside $\mathcal{A}_{(\gamma,i_x)}^M$, there is a (no-op) transition $q_1 \xrightarrow{a,q_1} q_2$
    * For each push/pop transition $q_1 \xrightarrow{a,\sigma} q_2$ inside $\mathcal{A}_{(\gamma,i_x)}^M$, there is a (push/pop resp.) transition $q_1 \xrightarrow{a,q_1,\sigma} q_2$

- For each state q_1, q_2 in $\bigcup_{i_x} N_{(\gamma,i_x)}$ and each no-op transition $q_2 \xrightarrow{a} q_3$ in the constituent automaton there is a transition $q_1 \xrightarrow{a,q_2} q_3$. 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.

9.2.2.2 $\mathbf{let}\, x^\beta = N \,\mathbf{in}\, M$

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

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

9.3 Reduction from SVPCMA

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

This proof will use many similar to the ideas we already presented in Section 8.1 in how RML terms can model VASS. In particular, we will use a $\overset{\curvearrowleft}{q_1 \quad a_1}$-thread to represent a data value, and a $\overset{\curvearrowleft}{q_2 \quad q_*}$-pair to represent a push-move, whilst the automaton state is simply stored in a global variable.

To reduce SVPCMA emptiness to observational equivalence of $\text{RML}_{\text{VPCMA}}$-terms, we will first alter the given SVPCMA to make the reduction to RML-terms easier. We already saw, in Chapter 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. This will mean that, like in the VASS-coverability reduction in Section 8.1, the two terms we will test for observational equivalence can differ just by an assert statement, asserting that a final state has not been reached.

How then, do we construct the RML terms from $\mathcal{A}'$? As mentioned above, each data value will be represented by a $\overset{\curvearrowleft}{q_1 \; a_1}$-thread. Hence, a transition reading a new data value will be represented by O playing q_1. The class memory function's value for this data value is then stored in a variable local to q_1. 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 in a variable local to the q_2-move. As we saw in Section 8.1, well-bracketing forces these "pushes" to be popped (by the moves a_* and a_2) in the correct order. Obviously by the above, if there is a push-move taking a fresh data value, the $\overset{\curvearrowleft}{q_1 \; q_*}$-thread must be created first, and then immediately followed by the $\overset{\curvearrowleft}{q_2 \; q_*}$-moves. Furthermore, as we saw in Section 9.2, 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 - as both the automaton and input are non-deterministic. We will do this by alternating O's plays between those which correspond to transitions of the SVPCMA as described already, and a simple q_1-move that provides as int-input, which transition will be fired next. The term, using a global variable, can keep track of whether the next O-move should be providing input, or simulating a transition.

Using these ideas, we will prove the following result:

Proposition 9.3.1. *Given a weak SVPCMA such that the automaton can only arrive at a final state with an empty stack, there are* RML *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.

```
1   let
2     State = ref ⟦q0⟧
3     Next_Move = ref ⟦Unknown⟧
4   in
5     λx^int.
6       let
7         CMF = ref −
8       in
9         assert* (!State ∉ ⟦F_G⟧);
10        if (!Next_Move == ⟦Unknown⟧) then
11          assert (x ∈ ⟦Δ⟧); Next_Move := x;
12          if (!Next_Move ∈ ⟦Δ_push ∩ Δ_fresh⟧) then CMF := ⟦⊥⟧
13          else skip
14        else
15          assert (!Next_Move ∈ ⟦Δ_noop ∩ Δ_fresh⟧);
16          ⋃_{(q,⊥,σ,q')∈Δ_noop∩Δ_fresh} :
17            if (!Next_Move == ⟦(q,⊥,σ,q')⟧) then
18              assert (!State == ⟦q⟧); State := ⟦q'⟧; CMF := q';
19            else skip
20          Next_Move := ⟦Unknown⟧
21        λf^{unit→unit}.
22          if (!Next_Move ∈ ⟦Δ_noop ∩ Δ_old⟧) then
23            ⋃_{(q,s,σ,q')∈Δ_noop∩Δ_old} :
24              if (!Next_Move == ⟦(q,s,σ,q')⟧) then
25                assert (!State == ⟦q⟧); assert (!CMF == ⟦s⟧);
26                State := ⟦q'⟧; CMF := q'
27              else skip
28            Next_Move := ⟦Unknown⟧
29          else if (!Next_Move ∈ ⟦Δ_push⟧) then
30            let
31              Stack_Letter = ref −
32            in
33              ⋃_{(q,s,σ,γ,q')∈Δ_push} :
34                if (!Next_Move == ⟦(q,s,σ,γ,q')⟧) then
35                  assert (!State == ⟦q⟧); assert (!CMF == ⟦s⟧);
36                  State := ⟦q'⟧; CMF := q'; Stack_Letter := ⟦γ⟧
37                else skip
38              Next_Move := ⟦Unknown⟧
39              f(skip);
40              assert (!Next_Move ∈ ⟦Δ_pop⟧);
41              ⋃_{(q,s,σ,γ,q')∈Δ_pop} :
42                if (!Next_Move == ⟦(q,s,σ,γ,q')⟧) then
43                  assert (!State == ⟦q⟧); assert (!CMF == ⟦s⟧);
44                  assert (!Stack_Letter == ⟦γ⟧;
45                  State := ⟦q'⟧; CMF := q'
46                else skip
47              Next_Move := ⟦Unknown⟧
48          else
49            assert False
```

Figure 9.2: The term encoding SVPCMA emptiness

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. As with the reductions in Chapter 8, we will 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 $[\![-]\!] : \Gamma \to \mathbb{N}$.

The term is shown in Fig. 9.2, and we now discuss how it simulates an SVPCMA. The term begins by setting up two global variables:

- `State`. This variable will keep track of the state the automaton is in, taking values from $[\![Q]\!]$; and
- `Next_Move`. This variable 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 after a transition is followed, 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 $\overleftarrow{q_1 \ a_1}$-moves. The local variable for $\overleftarrow{q_1 \ a_1}$-threads, `CMF`, is then setup. This will take a value from $[\![Q]\!]$, to represent the class memory function's memory of the data value this thread corresponds to.

As discussed above, there are two ways in which a $\overleftarrow{q_1 \ a_1}$-thread can be played: either letting O provide input as to which transition will be followed next, or to represent a transition in Δ_{fresh}. Which form is used is determined by whether or not the global variable `Next_Move` is set to $[\![Unknown]\!]$. If it is, then this play is taken to be giving us the next move to take. In this case the input O provided to the λ-abstraction is taken to be the integer corresponding to the next move under the $[\![-]\!]$-mapping, and it is stored to the global variable ready for the next O-move. Otherwise, we assume that these $\overleftarrow{q_1 \ a_1}$-moves are representing a noop-move taking a

fresh data value. The state is updated according to the transition being taken (stored in the `Next_Move` variable), and the local `CMF` variable is set to the new current state.

Transitions in $\Delta_{push} \cap \Delta_{fresh}$ are handled in a slightly complicated manner: when O provides the input that the next transition to be made will be such a transition, that $\overset{\curvearrowleft}{q_1\ a_1}$-thread will be used to represent the data value, and the next O-move will correspond to the push-move. In this case the variable `CMF` is used, and set to $[\![\bot]\!]$, which no other thread can have at this point, so it must be used for the next push move.

We note that we also included, after setting up the variable `CMF`, the assert statement that differs between the two terms: as after every transition this term simulates there will be a q_1-move "asking for the next move", it is a good place to insert the check of whether a final configuration has been reached. The assert statement will only fail if such a configuration is reached, and hence the strategies for the two terms will only be different if a final state is reachable by some O-input.

We now look at second λ-abstraction level. This corresponds to O making a q_2-move. As previously discussed, there are two possibilities for q_2-moves: either a push-move is being made (and so P must follow the q_2-move up with a q_*), or a noop-move is made, reading an existing data value. In this latter case the first branch of the **if** is taken, and the changes to the existing variables are made as expected.

If a push-move is being made, 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(**skip**), 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. Once P plays q_*, O is free to begin new threads and continue play as before, "within" the push. Once this instance of f(**skip**) is fully evaluated, O plays the move a_*. At this point the pop move corresponding to this push is simulated. 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 push-move, and so the same data value that was used in the push move must be used in the pop-move. □

In the above we have used an int-type, to make it easy for O to "choose" the next transition to be fired. We note that we could have used only unit-types, using the same trick as used in Section 8.1.1 to avoid the int-type. Hence, it's straightforward to adjust the above to show that we can reduce emptiness of SVPCMA to any type of the form $\vdash \beta \to (\beta \to \beta) \to \beta$. This completes the proof of Theorem 9.0.1.

9.4 Summary

We began this chapter by noting that the only RML terms without free variables for which decidability of observational equivalence is unknown are those of a type of the form $\beta \to (\beta \to \ldots \beta) \to \beta$. We chose $\text{RML}_{\text{VPCMA}}$ to include these types and, gave a reduction from $\text{RML}_{\text{VPCMA}}$ terms to deterministic weak SVPCMA recognising the term's strategy. Hence, observational equivalence of $\text{RML}_{\text{VPCMA}}$ terms is reducible to emptiness of VPCMA.

We then were able to reduce emptiness of VPCMA to observational equivalence at type $\vdash \beta \to (\beta \to \beta) \to \beta$ – the simplest type in $\text{RML}_{\text{VPCMA}}$ and not in an already known-decidable fragment. Thus we get the result that observational equivalence of $\text{RML}_{\text{VPCMA}}$ terms is decidable if and only if VPCMA emptiness is.

Unfortunately, the decidability of VPCMA emptiness is an open question, equivalent to whether reachability for extended BVASS is decidable. The decidability of (normal) BVASS reachability has been an open question for over a decade ([21]), and hence it is beyond the scope of this thesis to attempt to resolve the decidability of EBVASS reachability.

Chapter 10

Conclusion

This thesis has presented results both in automata theory and in algorithmic game semantics. In Part I we explored new forms of class memory automata, and studied their algorithmic properties. In Part II we used these new forms of class memory automata to represent terms of the call-by-value language with state RML. Due to a full-abstraction result, observational equivalence of RML terms was thus reduced to equivalence of these automata. By the results in Part I we were thus able to obtain decidability results for new fragments of RML.

10.1 Class memory automata

Class memory automata were introduced [14] as a conceptually cleaner equivalent presentation of data automata. CMA operate over an infinite alphabet by remembering a finite bit of information for each data value seen in the run. Acceptance at the end of the run can be contingent on the stored information for each data value.

In Chapter 3 we relaxed this acceptance condition, and studied the resultant *weak* CMA. We found they had better closure and algorithmic properties, and were able to show that they were equivalent to several distinct forms of automata already presented in the literature[56, 90, 22]. In Chapter 4 we added a tree structure to the data values, and presented a form of CMA over these nested data values. We found that these CMA were undecidable in general, but with the weakness constraint decidability was recovered. In Chapter 5 we investigated a form of visibly pushdown CMA, in which data values could be added to the stack to ensure that corresponding push- and pop-moves had the same data value. We showed that emptiness for these VPCMA is

equivalent to reachability of extended branching VASS, for which decidability is an open question. Importantly, for CMA, NDCMA, and VPCMA, we found that in the deterministic weak case the automata were closed under all Boolean operations, a potentially valuable feature in modelling systems.

Future work. It is obvious that the open question of EBVASS decidability is an important one for these results, and it will also have significance for the results in Part II of this thesis.

More generally, finding classes of automata over infinite alphabets with a decidable equivalence problem is potentially very helpful in verifying properties of systems by modelling their behaviour, and this work has suggested there may be many more forms of CMA with this property. In particular, the work on NDCMA raises interesting questions about what possible structure can be given to the data values whilst still retaining decidability.

Separately, we defined a notion of visibly pushdown CMA, but this was not the simplest possible definition, as we enforced that corresponding push- and pop-moves have the same data value. There are potentially cases where this condition may not be required, and whether the relaxation of this condition would affect complexity has not been explored. In [17, 66] forms of (non-visibly) pushdown automata with registers have been studied and found to have decidable algorithmic properties – future work could investigate whether pushdown automata with a class memory function can yield a decidable form of automaton.

10.2 RML

RML is a call-by-value higher-order language with state, which may be viewed as a call-by-value version of Reynold's Idealized Algol [82]. It is closely related to ML [35], restricted to ground-type references and with a bad-variable constructor. Game semantics for RML provides us with a method for testing observational equivalence of RML terms by reducing the problem to equivalence of strategies for those terms. We gave algorithms for constructing (a form of) CMA for RML terms of certain types, thus reducing observational equivalence to equivalence of these CMA. By this method, we were able to decide two fragments of RML, containing all first-order terms with certain free variables. We also showed that a fragment containing a

second-order type, $\vdash \beta \to (\beta \to \beta) \to \beta$, is equivalent to emptiness of VPCMA (and hence EBVASS reachability).

Table 10.1: Summary of decidability results for RML.
A reference of [†] indicates the result is presented in this thesis. A complexity of $\perp$ indicates undecidability.

Fragment	Representative Type Sequent	Complexity & Ref.
	Decidable	
O-Strict / $\text{RML}_{\text{O-Str}}$	$((\beta \to \cdots \to \beta) \to \beta) \to \cdots \to \beta \vdash$ $(\beta \to \cdots \to \beta) \to \beta$	ExpTime [41, 40]
O-Strict + Recursion	as $\text{RML}_{\text{O-Str}}$, with $\beta \to \beta$ recursion	DPDA-Hard [40]
RML_{CMA}	$(\beta \to \beta) \to \cdots \to (\beta \to \beta) \to \beta \vdash \beta \to \beta \to \beta$	ExpSpace-hard [†] in 2-ExpSpace [40]
$\text{RML}^{\text{P-Str}}_{2\vdash 1}$	$(\beta \to \cdots \to \beta) \to \beta \vdash \beta \to \cdots \to \beta$	Ackermann Hard [18], [†]
$\text{RML}^{\text{LHS-O-str}}_{2\vdash 1}$	$(\beta \to \beta) \to \cdots \to (\beta \to \beta) \to \beta \vdash$ $\beta \to \cdots \to \beta$	Ackermann Hard [18], [†]
	Unknown	
$\text{RML}_{\text{VPCMA}}$	$(\beta \to \beta) \to \cdots \to (\beta \to \beta) \to \beta \vdash$ $\beta \to (\beta \to \cdots \to \beta) \to \beta$	EBVASS-reachability complete [†]
	Undecidable	
Third-Order	$\vdash ((\beta \to \beta) \to \beta) \to \beta$ $(((\beta \to \beta) \to \beta) \to \beta) \to \beta \vdash \beta$	$\perp$ [40, 18]
Second-Order	$\vdash (\beta \to \beta) \to \beta \to \beta$ $((\beta \to \beta) \to \beta \to \beta) \to \beta \vdash \beta$	$\perp$ [40, 18]
Recursion	recursion of type $(\beta \to \beta) \to \beta$	$\perp$ [40, 18]
Second-Order	$\vdash \beta \to \beta \to (\beta \to \beta) \to \beta$ $(\beta \to \beta \to (\beta \to \beta) \to \beta) \to \beta \vdash \beta$	$\perp$ [†]
Free variables	$(\beta \to \beta \to \beta \to \beta) \to \beta \to \beta \to \beta \vdash \beta$	$\perp$ [†]
	Open	
RML_{CMA} + Recursion	as RML_{CMA}, with $\beta \to \beta$ recursion	-
Free variables	$(\beta \to \cdots \to \beta) \to (\beta \to \cdots \to \beta) \to \beta \vdash \beta$ $(\beta \to \beta \to \beta) \to \cdots \to (\beta \to \beta \to \beta) \to \beta \vdash \beta$	-
Combining LHS + RHS	$((\beta \to \beta) \to \beta) \to \beta \vdash \beta \to \beta \to \beta$ $(\beta \to \cdots \to \beta) \to \beta \vdash (\beta \to \beta) \to \beta$	-

In addition, we showed several hardness and undecidability results. By reduction from VASS coverability, we showed that RML_{CMA}, for which OE was first shown decidable in [40], is ExpSpace-hard. We also reduced reset VASS coverability to OE of terms of type $\vdash \textsf{unit} \to \textsf{unit} \to \textsf{unit} \to \textsf{unit}$, showing that observational equivalence for the fragments decided in Chapter 7 is non-primitive-recursive. By reduction from

reset VASS reachability, we showed terms of type $\vdash \beta \to \beta \to (\beta \to \beta) \to \beta$ to have an undecidable observational equivalence problem. This proof easily generalises to any order 2-term with arity 3 or greater and a final argument of order 1. In particular, together with a result from [40, 18] this reduces the potentially decidable order 2 types to just those of the shape $\vdash (\beta \to \cdots \to \beta) \to \beta$ (in $\mathrm{RML}_{\text{O-Str}}$) or $\vdash \beta \to (\beta \to \cdots \to \beta) \to \beta$ (in $\mathrm{RML}_{\mathrm{VPCMA}}$). In Section 8.2, we obtained the first undecidability result for RML using just a second-order free variable. These results are summarised in Table 10.1.

In particular, note that for all RHS types we either have a result showing decidability or undecidability of a fragment containing that type, with the exception of the types in $\mathrm{RML}_{\mathrm{VPCMA}}$, for which we know OE is equivalent to EBVASS reachability. However, we do not yet have a complete classification of which LHS types give undecidability or decidability, nor a complete picture of which combinations of LHS and RHS types remain decidable. In the rest of this section we discuss these open cases.

10.2.1 Open cases

Free variables. In [40] Hopkins showed that any 4th-order free variable leads to undecidability. $\mathrm{RML}_{\text{O-Str}}$ permits 3rd-order free variables in which each argument is of arity 1, and so these do not lead to undecidability. From results in Section 8.1 together with results from [40] we get that 3rd-order free variables in which the arguments have arity 3 or greater lead to undecidability. When the arguments have arity 2, [40] showed that if the first argument is order 1 then we get undecidability, and if only the second argument is order 1 we are in the case covered by $\mathrm{RML}_{\mathrm{VPCMA}}$, and so equivalent to EBVASS reachability.

This leaves 2nd-order free variables. Where the arguments are arity 1, decidability descends from the 3rd-order case. In Section 7.2 we showed decidability for all 2nd-order free variables of arity 1. However in Section 8.2 we showed undecidability with a second-order type of arity 3 in which the first argument had arity 3.

This leaves open where the divide between decidability and undecidability for 2nd-order free variables lies. In particular, do all 2nd-order free variables of arity 2 maintain decidability? And is decidability retained with 2nd-order variables of

unlimited arity if each argument has arity at most 2? These types are shown in penultimate row of Table 10.1

The issue in deciding these – and was used in the undecidability argument for 2nd-order free variables – is how plays in the arguments are interleaved. It seems possible that by efficient encoding of this information, free variables with arguments of arity no more than 2 may be encoded using a kind of NDCMA, using just one level of data values for each argument.

Combining permitted LHS and RHS types. In the decidability of RHS types we have been able to choose which free variables we permit to help our inductive constructions proceed naturally. This leaves open whether, given a RHS type θ and LHS type θ' both of which we know to be decidable in some contexts, whether terms of type $x : \theta' \vdash M : \theta$ have decidable OE.

In particular, as part of $\mathrm{RML}_{\text{O-Str}}$ we know that free variables of type $((\beta \to \cdots \to \beta) \to \beta) \to \beta$ can be decidable, and from $\mathrm{RML}_{\text{CMA}}$ we know terms of type $\beta \to \beta \to \beta$ can be decidable. We do not know whether terms of type $x : ((\beta \to \cdots \to \beta) \to \beta) \to \beta \vdash M : \beta \to \beta \to \beta$ are decidable. However, it seems plausible that it would be possible to reduce these terms to VPCMA or similar automata, with the visibly-pushdown aspect of the automata being used for dealing with the LHS, and the CMA used to keep track of threads on the RHS.

If the type on the RHS were extended to $\beta \to \beta \to \beta \to \beta$ this would seem to require a kind of visibly pushdown NDCMA. However, such automata are unlikely to be decidable, as the undecidable type $\vdash \beta \to \beta \to (\beta \to \beta) \to \beta$ could likely be reduced to equivalence of such automata. On the other hand, we currently have no arguments that this type would be undecidable, and such an argument would likely have to rely on the complex interactions in plays between LHS and RHS types.

Similarly, we may wish to consider terms of types of the form $(\beta \to \cdots \to \beta) \to \beta \vdash (\beta \to \cdots \to \beta) \to \beta$, where the LHS is in $\mathrm{RML}^{\text{P-Str}}_{2\vdash 1}$, and the RHS is in $\mathrm{RML}_{\text{O-Str}}$. Again, this would seem to require – if we wished to use the same methods as in this thesis – a kind of visibly pushdown NDCMA, which is likely undecidable.

Recursion. We have not, in this thesis, looked at recursion beyond simple while-loops. In [40] Hopkins showed $\mathrm{RML}_{\text{O-Str}}$ was decidable with simple $\beta \to \beta$ recursion, while $(\beta \to \beta) \to \beta$ immediately led to undecidability. It remains unclear whether

decidability can be retained for $\mathrm{RML}_{\mathrm{CMA}}$ in the presence of $\beta \to \beta$ recursion, and if so whether this can easily be extended to $\beta \to \beta \to \beta$. Hopkins suggests that a kind of non-visibly pushdown CMA may be suitable for deciding this fragment – if a decidable such form of CMA exists.

If this proves decidable the next question is whether recursion can be introduced to types $\vdash \beta \to \cdots \to \beta$ or, if EBVASS reachability is decidable, $\vdash \beta \to (\beta \to \beta) \to \beta$, whilst retaining decidability.

Bibliography

[1] S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In *LICS*, pages 334–344. IEEE Computer Society, 1998.

[2] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Inf. Comput.*, 163(2):409–470, 2000.

[3] S. Abramsky, P. Malacaria, and R. Jagadeesan. Full abstraction for PCF. In *TACS*, volume 789 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1994.

[4] S. Abramsky and G. McCusker. Games for recursive types. In *Theory and Formal Methods*, pages 1–20. Imperial College Press, 1994.

[5] S. Abramsky and G. McCusker. Linearity, sharing and state: a fully abstract game semantics for idealized algol with active expressions. *Electr. Notes Theor. Comput. Sci.*, 3:2–14, 1996.

[6] S. Abramsky and G. McCusker. Call-by-value games. In *CSL*, volume 1414 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1997.

[7] S. Abramsky and G. McCusker. Full abstraction for idealized algol with passive expressions. *Theor. Comput. Sci.*, 227(1-2):3–42, 1999.

[8] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *POPL*, pages 340–353. ACM, 2009.

[9] R. Alur, L. D'Antoni, J. V. Deshmukh, M. Raghothaman, and Y. Yuan. Regular functions and cost register automata. In *LICS*, pages 13–22. IEEE Computer Society, 2013.

[10] R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, pages 202–211. ACM, 2004.

[11] T. Araki and T. Kasami. Some decision problems related to the reachability problem for petri nets. *Theor. Comput. Sci.*, 3(1):85–104, 1976.

[12] M. Arenas, W. Fan, and L. Libkin. Consistency of XML specifications. In *Inconsistency Tolerance*, volume 3300 of *Lecture Notes in Computer Science*, pages 15–41. Springer, 2005.

[13] H. Björklund and M. Bojańczyk. Shuffle expressions and words with nested data. In *MFCS*, volume 4708 of *Lecture Notes in Computer Science*, pages 750–761. Springer, 2007.

[14] H. Björklund and T. Schwentick. On notions of regularity for data languages. In *FCT*, volume 4639 of *Lecture Notes in Computer Science*, pages 88–99. Springer, 2007.

[15] H. Björklund and T. Schwentick. On notions of regularity for data languages. *Theor. Comput. Sci.*, 411(4-5):702–715, 2010.

[16] M. Bojańczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David. Two-variable logic on words with data. In *LICS*, pages 7–16. IEEE Computer Society, 2006.

[17] E. Y. C. Cheng and M. Kaminski. Context-free languages over infinite alphabets. *Acta Inf.*, 35(3):245–267, 1998.

[18] C. Cotton-Barratt, D. Hopkins, A. S. Murawski, and C. L. Ong. Fragments of ML decidable by nested data class memory automata. In *FoSSaCS*, volume 9034 of *Lecture Notes in Computer Science*, pages 249–263. Springer, 2015.

[19] C. Cotton-Barratt, A. S. Murawski, and C. L. Ong. Weak and nested class memory automata. In *LATA*, volume 8977 of *Lecture Notes in Computer Science*, pages 188–199. Springer, 2015.

[20] C. David, L. Libkin, and T. Tan. On the satisfiability of two-variable logic over data words. In *LPAR (Yogyakarta)*, volume 6397 of *Lecture Notes in Computer Science*, pages 248–262. Springer, 2010.

[21] P. de Groote, B. Guillaume, and S. Salvati. Vector addition tree automata. In *LICS*, pages 64–73. IEEE Computer Society, 2004.

[22] N. Decker, P. Habermehl, M. Leucker, and D. Thoma. Ordered navigation on multi-attributed data words. In *CONCUR*, volume 8704 of *Lecture Notes in Computer Science*, pages 497–511. Springer, 2014.

[23] N. Decker and D. Thoma. On freeze LTL with ordered attributes. In *FoSSaCS*, volume 9634 of *Lecture Notes in Computer Science*, pages 269–284. Springer, 2016.

[24] S. Demri and R. Lazic. LTL with the freeze quantifier and register automata. In *LICS*, pages 17–26. IEEE Computer Society, 2006.

[25] S. Demri, R. Lazic, and D. Nowak. On the freeze quantifier in constraint LTL: decidability and complexity. In *TIME*, pages 113–121. IEEE Computer Society, 2005.

[26] D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *ICFP*, pages 143–156. ACM, 2010.

[27] C. Dufourd, A. Finkel, and P. Schnoebelen. Reset nets between decidability and undecidability. In *ICALP*, volume 1443 of *Lecture Notes in Computer Science*, pages 103–115. Springer, 1998.

[28] J. Esparza and M. Nielsen. Decidability issues for petri nets - a survey. *Bulletin of the EATCS*, 52:244–262, 1994.

[29] A. Finkel. A generalization of the procedure of karp and miller to well structured transition systems. In *ICALP*, volume 267 of *Lecture Notes in Computer Science*, pages 499–508. Springer, 1987.

[30] A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.

[31] D. R. Ghica. Regular-language semantics for a call-by-value programming language. *Electr. Notes Theor. Comput. Sci.*, 45:106–118, 2001.

[32] D. R. Ghica and G. McCusker. Reasoning about idealized ALGOL using regular languages. In *ICALP*, volume 1853 of *Lecture Notes in Computer Science*, pages 103–115. Springer, 2000.

[33] J. L. Gischer. Shuffle languages, petri nets, and context-sensitive grammars. *Commun. ACM*, 24(9):597–605, 1981.

[34] B. Godlin and O. Strichman. Regression verification. In *DAC*, pages 466–471. ACM, 2009.

[35] M. J. C. Gordon, R. Milner, L. Morris, M. C. Newey, and C. P. Wadsworth. A metalanguage for interactive proof in LCF. In *POPL*, pages 119–130. ACM Press, 1978.

[36] E. Grädel and M. Otto. On logics with two variables. *Theor. Comput. Sci.*, 224(1-2):73–113, 1999.

[37] O. Grumberg, O. Kupferman, and S. Sheinvald. Variable automata over infinite alphabets. In *LATA*, volume 6031 of *Lecture Notes in Computer Science*, pages 561–572. Springer, 2010.

[38] K. Honda and N. Yoshida. Game-theoretic analysis of call-by-value computation. *Theor. Comput. Sci.*, 221(1-2):393–456, 1999.

[39] J. E. Hopcroft and J. Pansiot. On the reachability problem for 5-dimensional vector addition systems. *Theor. Comput. Sci.*, 8:135–159, 1979.

[40] D. Hopkins. *Game Semantics Based Equivalence Checking of Higher-Order Programs.* PhD thesis, Department of Computer Science, University of Oxford, 2012.

[41] D. Hopkins, A. S. Murawski, and C. L. Ong. A fragment of ML decidable by visibly pushdown automata. In *ICALP (2)*, volume 6756 of *Lecture Notes in Computer Science*, pages 149–161. Springer, 2011.

[42] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for pcf: I, ii and iii (preliminary version). Technical report, 1994.

[43] J. M. E. Hyland and C. L. Ong. On full abstraction for PCF: i, ii, and III. *Inf. Comput.*, 163(2):285–408, 2000.

[44] Y. Ishihara, H. Kuwada, and T. Fujiwara. The absolute consistency problem of XML schema mappings with data values between restricted dtds. In *DEXA (1)*, volume 8644 of *Lecture Notes in Computer Science*, pages 317–327. Springer, 2014.

[45] F. Jacquemard, L. Segoufin, and J. Dimino. Fo2(<, +1, ˜) on data trees, data tree automata and branching vector addition systems. *Logical Methods in Computer Science*, 12(2), 2016.

[46] G. Kahn. Natural semantics. In *STACS*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[47] M. Kaminski and N. Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.

[48] A. Kara, T. Schwentick, and T. Tan. Feasible automata for two-variable logic with successor on data words. In *LATA*, volume 7183 of *Lecture Notes in Computer Science*, pages 351–362. Springer, 2012.

[49] S. R. Kosaraju. Decidability of reachability in vector addition systems (preliminary version). In *STOC*, pages 267–281. ACM, 1982.

[50] V. Koutavas and M. Wand. Small bisimulations for reasoning about higher-order imperative programs. In *POPL*, pages 141–152. ACM, 2006.

[51] D. Kozen. *Automata and computability.* Undergraduate texts in computer science. Springer, 1997.

[52] J. Laird. Full abstraction for functional languages with control. In *LICS*, pages 58–67. IEEE Computer Society, 1997.

[53] R. Lazic. The reachability problem for branching vector addition systems requires doubly-exponential space. *Inf. Process. Lett.*, 110(17):740–745, 2010.

[54] R. J. Lipton. *The reachability problem requires exponential space*. Research report (Yale University. Department of Computer Science). Department of Computer Science, Yale University, 1976.

[55] R. Loader. Finitary PCF is not decidable. *Theor. Comput. Sci.*, 266(1-2):341–364, 2001.

[56] A. Manuel and R. Ramanujam. Counting multiplicity over infinite alphabets. In *RP*, volume 5797 of *Lecture Notes in Computer Science*, pages 141–153. Springer, 2009.

[57] E. W. Mayr. An algorithm for the general petri net reachability problem. *SIAM J. Comput.*, 13(3):441–460, 1984.

[58] G. McCusker. Games and definability for FPC. *Bulletin of Symbolic Logic*, 3(3):347–362, 1997.

[59] R. Meyer. On boundedness in depth in the pi-calculus. In *IFIP TCS*, volume 273 of *IFIP*, pages 477–489. Springer, 2008.

[60] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[61] A. S. Murawski. On program equivalence in languages with ground-type references. In *LICS*, page 108. IEEE Computer Society, 2003.

[62] A. S. Murawski. Functions with local state: from regularity to undecidability. In *GALOP*, pages 124–138, 2005.

[63] A. S. Murawski. Games for complexity of second-order call-by-name programs. *Theor. Comput. Sci.*, 343(1-2):207–236, 2005.

[64] A. S. Murawski. Bad variables under control. In *CSL*, volume 4646 of *Lecture Notes in Computer Science*, pages 558–572. Springer, 2007.

[65] A. S. Murawski, C. L. Ong, and I. Walukiewicz. Idealized algol with ground recursion, and DPDA equivalence. In *ICALP*, volume 3580 of *Lecture Notes in Computer Science*, pages 917–929. Springer, 2005.

[66] A. S. Murawski, S. J. Ramsay, and N. Tzevelekos. Reachability in pushdown register automata. In *MFCS (1)*, volume 8634 of *Lecture Notes in Computer Science*, pages 464–473. Springer, 2014.

[67] A. S. Murawski and N. Tzevelekos. Full abstraction for reduced ML. In *FOSSACS*, volume 5504 of *Lecture Notes in Computer Science*, pages 32–47. Springer, 2009.

[68] A. S. Murawski and N. Tzevelekos. Algorithmic nominal game semantics. In *ESOP*, volume 6602 of *Lecture Notes in Computer Science*, pages 419–438. Springer, 2011.

[69] A. S. Murawski and N. Tzevelekos. Game semantics for good general references. In *LICS*, pages 75–84. IEEE Computer Society, 2011.

[70] A. S. Murawski and N. Tzevelekos. Algorithmic games for full ground references. In *ICALP (2)*, volume 7392 of *Lecture Notes in Computer Science*, pages 312–324. Springer, 2012.

[71] A. S. Murawski and N. Tzevelekos. Game semantics for interface middleweight java. In *POPL*, pages 517–528. ACM, 2014.

[72] A. S. Murawski and N. Tzevelekos. Game semantics for nominal exceptions. In *FoSSaCS*, volume 8412 of *Lecture Notes in Computer Science*, pages 164–179. Springer, 2014.

[73] A. S. Murawski and I. Walukiewicz. Third-order idealized algol with iteration is decidable. In *FoSSaCS*, volume 3441 of *Lecture Notes in Computer Science*, pages 202–218. Springer, 2005.

[74] F. Neven and T. Schwentick. Xpath containment in the presence of disjunction, dtds, and variables. In *ICDT*, volume 2572 of *Lecture Notes in Computer Science*, pages 312–326. Springer, 2003.

[75] F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.*, 5(3):403–435, 2004.

[76] H. Nickau. Hereditarily sequential functionals. In *LFCS*, volume 813 of *Lecture Notes in Computer Science*, pages 253–264. Springer, 1994.

[77] C. L. Ong. An approach to deciding the observational equivalence of algol-like languages. *Ann. Pure Appl. Logic*, 130(1-3):125–171, 2004.

[78] C. A. Petri. Fundamentals of a theory of asynchronous information flow. In *IFIP Congress*, pages 386–390, 1962.

[79] A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In *Higher order operational techniques in semantics*, pages 227–273. Cambridge University Press, 1998.

[80] G. D. Plotkin. A structural approach to operational semantics, 1981.

[81] C. Rackoff. The covering and boundedness problems for vector addition systems. *Theor. Comput. Sci.*, 6:223–231, 1978.

[82] J. C. Reynolds. The essence of ALGOL. In *Proceedings of the International Symposium on Algorithmic Languages*. Elsevier Science Inc., 1981.

[83] S. Schmitz. The complexity of reachability in vector addition systems. *SIGLOG News*, 3(1):4–21, 2016.

[84] P. Schnoebelen. Revisiting ackermann-hardness for lossy counter machines and reset petri nets. In *MFCS*, volume 6281 of *Lecture Notes in Computer Science*, pages 616–628. Springer, 2010.

[85] T. Schwentick. Xpath query containment. *SIGMOD Record*, 33(1):101–109, 2004.

[86] D. Scott. Outline of a mathematical theory of computation. Technical Report PRG02, OUCL, November 1970.

[87] D. S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. Technical Report PRG06, OUCL, August 1971.

[88] I. Stark. *Names and Higher-Order Functions.* PhD thesis, University of Cambridge, Dec. 1994.

[89] N. Tzevelekos. Fresh-register automata. In *POPL*, pages 295–306. ACM, 2011.

[90] N. Tzevelekos and R. Grigore. History-register automata. In *FoSSaCS*, volume 7794 of *Lecture Notes in Computer Science*, pages 17–33. Springer, 2013.

Index

$\perp$, 23

Class counting automata, 31
Class memory automata, 23
  Nested data, 43, 45, 102
  Scoping visibly pushdown, 68
  Signature, 23
  Visibly pushdown, 56
  Weak, 27
  with Labels, 25
Class memory function, 23
Closure properties
  of CMA, 25, 38
  of NDCMA, 46
  of VPCMA, 59
  of weak CMA, 28

Data automata, 21
  locally prefix-closed, 30
  Nested, 51
  Weak, 33
Data class, 21
Data values, 20
  Nested, 42
Data words, 20
Decidability
  of $\mathrm{RML}_{2\vdash 1}^{\mathrm{LHS-O-str}}$, 121
  of $\mathrm{RML}_{2\vdash 1}^{\text{P-Str}}$, 106
  of CMA, 25
  of Idealized Algol, 11
  of weak CMA, 30
  of weak NDCMA, 49

Full abstraction, 10

Game semantics, 10, 83
  $\Rightarrow$, 87
  $\otimes$, 87
  $\rightarrow$, 88
  Arena, 86
  Arenas for RML types, 87
  Denotation of RML terms, 88
  Justification pointer, 84
  Justified sequence, 84
  Play, 85
  Prearena, 84
  Prearenas for RML terms, 88
  Strategy, 86

Hardness
  of $\mathrm{RML}_{2\vdash 1}^{\mathrm{LHS-O-str}}$, 134
  of $\mathrm{RML}_{\mathrm{CMA}}$, 130
  of $\mathrm{RML}_{2\vdash 1}^{\text{P-Str}}$, 134
  of weak NDCMA, 47
History register automata, 32

Idealized Algol, 10

Logic on data words, 34
  $+1$, 35
  $\oplus 1$, 35
  Existentially bounded fragment, 35

Observational equivalence, 91

Prearena, 84
Pushdown alphabet, 19

Queue machines, 144

Reduced ML, 10, 93
RML, 10, 79
  $\mathrm{RML}_{2\vdash 1}$, 99
  $\mathrm{RML}_{2\vdash 1}^{\mathrm{LHS-O-str}}$, 119
  $\mathrm{RML}_{\mathrm{CMA}}$, 96
  $\mathrm{RML}_{\mathrm{O\text{-}Str}}$, 93
  $\mathrm{RML}_{2\vdash 1}^{\mathrm{P\text{-}Str}}$, 102
  $\mathrm{RML}_{\mathrm{VPCMA}}$, 156
  Canonical form, 93
  Finitary, 79
  Operational semantics, 81
  store, 81

Strategy, 86

Type, 79
Types, 79
  Arity, 80
  Order, 80

Undecidability
  at types $(\beta \to \beta \to \beta \to \beta) \to \beta \to \beta \to \beta \vdash \beta$, 144
  at types $\vdash \beta \to \beta \to (\beta \to \beta) \to \beta$, 141
  of NDCMA, 47

VASS, 18
  Branching, 60
  Extended branching, 61
  with Resets, 19
Visibility, 85
Visibly pushdown automata, 19

Well-bracketing, 85
Well-structured transition systems, 48