Game Semantics Based Equivalence Checking of Higher-Order Programs

David Hopkins
St Hugh's College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy
Trinity 2012

Abstract

This thesis examines the use of game semantics for the automatic equivalence checking of higher-order programs. Game semantics has proved to be a powerful method for constructing fully abstract models of logics and programming languages. Furthermore, the concrete nature of the semantics lends itself to algorithmic analysis. The game-semantic model can be used to identify fragments of languages which have a decidable observational equivalence problem. We investigate decidability results for different languages as well as the efficiency of these algorithms in practice.

First we consider the call-by-value higher-order language with state, RML. This can be viewed as a canonical restriction of Standard ML to ground-type references. The O-strict fragment of RML is the largest set of type sequents for which, in the game-semantic denotation, justification pointers from O-moves are always uniquely reconstructible from the underlying move sequence. The O-strict fragment is surprisingly expressive, including higher-order types and difficult examples from the literature. By representing strategies as Visibly Pushdown Automata (VPA) we show that observational equivalence of O-strict terms is decidable (and in fact is ExpTime-complete). We then consider extensions of the O-strict fragment. Adding general recursion or using most non-O-strict types leads to undecidability. However, a limited form of recursion can be added while still preserving decidability (although the full power of DPDA is required).

Next we examine languages with non-local control. This involves adding **call/cc** to our language and is known to correspond to dropping the game-semantic bracketing condition. In the call-by-name game-semantic model of Idealized Algol (IA), in which answers cannot justify questions, the visibility condition still implies a form of weak bracketing. By making bracketing violations explicit we show that we can still model the entire third-order fragment using VPA.

We have also implemented tools based on these algorithms. Our model checkers Homer and Hector perform equivalence checking for third-order IA and O-strict RML respectively. Homer uses a naïve explicit state method whereas Hector takes advantage of on-the-fly model checking. Our tools perform well on small yet challenging examples. On negative instances, the on-the-fly approach allows Hector to outperform Homer. To improve their performance, we also consider using ideas from symbolic execution. We propose a representation for finite automata using transitions labelled with formulas and guards which aims to take advantage of the symmetries of the game-semantic model so that strategies can be represented compactly. We refer to this representation as Symbolically Executed Automata (SEA). Using SEA allows much larger data types to be handled but is not as effective on larger examples with small data types.

Acknowledgements

I would like to thank my supervisors Luke Ong and Andrzej Murawski for all the help and guidance they have provided me. Both have always freely given me the advice, encouragement and criticism I have needed from before I started my DPhil, through to the completion of this thesis.

I would also like to acknowledge Microsoft Research and a generous donation from Tony Hoare for funding this work.

Additionally, I would like to express my gratitude to my transfer and confirmation examiners Samson Abramsky, Daniel Kroening and Hongseok Yang for taking the time to read my work and providing me with valuable feedback and advice to help keep me on the right track.

I am grateful to my officemates for giving helpful advice, answering my questions and having insightful discussions, as well for as providing an enjoyable (if not always distraction-free) working environment.

My thanks also go to my friends both in Oxford and beyond who have provided a welcome relief from work. My housemates, the quiz team, my school friends and many others have helped keep me going.

Finally, I am incredibly thankful for everything my mum and dad have done for me. Without their love, advice and support (intellectual, emotional and financial) this work would never have happened.

Contents

Chapter 1

Introduction

1.1 Formal Verification

Computer programs have become an essential part of everyday life and so when they go wrong the effects can be disastrous. The size and complexity of modern programs makes it all too easy for bugs to be introduced during coding. Extensive testing can be used to help find and eradicate bugs. However, the number of possible inputs can be infeasibly large or even infinite and so testing cannot guarantee the complete absence of bugs. Formal verification seeks to address this problem by *proving* that a program behaves as it is designed to (or at least that it is guaranteed to satisfy some desirable properties — e.g. termination or not accessing unallocated storage). Verification is one of the most fundamental fields in computer science. In fact, Hoare proposes that the problem of developing a verifying compiler should be viewed as a "Grand Challenge" to the computer science community [45]. A challenging form of verification is equivalence checking [33, 43, 50, 59, 110]. This involves checking whether two programs have exactly the same behaviour. Equivalence actually subsumes many other interesting properties such as reachability or termination. It is particularly important when making optimisations (either automatically or manually) or when refactoring code, as we may wish to check that the optimised code will always behave the same as the original program. In reference to Hoare's "Grand Challenge", Godlin and Strichman suggest that "program equivalence can be thought of as a grand challenge in its own right" [44].

There are many known effective verification techniques. Static analysis and abstract interpretation can be used to efficiently over-approximate properties of programs [32]. Alternatively, model checking can be used to check whether a model of a system satisfies some property, often given as a temporal logic formula [29]. To increase scalability, various techniques such as symbolic model checking [68], bounded

model checking [18, 30], predicate abstraction [14, 15] and counterexample guided abstraction refinement [28] can be used. However, while these methods have been proved effective, they are all primarily aimed at verification of hardware or first-order imperative programs. The inclusion of higher-order functions in a language can introduce new challenges for verification techniques.

1.2 Higher-Order Programs

In higher-order functional languages such as Haskell or ML, functions are first-class objects which can be passed around. That is, higher-order functions which accept other functions as arguments are commonplace [27, 51, 85]. Higher-order languages tend to have strong type systems which can catch many bugs at compile time. Unfortunately, the type system will still let some bugs through. Various techniques for verifying safety properties of higher-order programs involve strengthening the type system [16, 19, 39, 109]. These tend not to be fully automatic and require manual assistance or annotation, although there are methods to minimise this by combining type theoretic approaches with techniques which have been successful for imperative programs such as predicate abstraction and abstract interpretation [53, 95].

Another approach to verifying higher-order programs is control-flow analysis [100, 69, 54]. Various forms of control-flow analysis exist for over-approximating the values which can appear at different program points. This can be very effective at finding bugs but as it is an over-approximation it may generate many spurious errors, even when the program is correct.

Recently, Higher-Order Recursion Schemes (HORS) have been proposed as a suitable formalism for verifying higher-order programs. Kobayashi shows how they can be used to encode pure functional programs over finite data types and suggests that HORS should be the target abstraction of CEGAR loops on higher-order programs [58]. HORS have a decidable modal μ-calculus and so many interesting properties can be checked [87]. The worst-case complexity of these algorithms is horrendous (n-ExpTime-complete) but experimental results suggest that in practice many interesting examples can be handled [57].

For proving equivalences in higher-order languages, logical relations have proved particularly effective [90, 101]. Recently, these have utilised step-indexing and other techniques to handle a wide range of different language features [9, 36]. However, such methods are not automatic and so require a large amount of proof by hand.

1.3 Game Semantics

An alternative method of verifying higher-order programs relies on game semantics. In recent years, game semantics has emerged as a powerful technique for modelling programming languages and logics. Game semantics views program execution as the playing of a two-player game between the program and its environment. It was used to give the first syntax independent fully abstract models of the pure functional prototypical programming language PCF [92, 3, 52, 83]. By making small changes to the rules of the game, fully abstract models of other languages could be constructed, including languages with state [6, 7], general references [2], recursive types [4, 66], non-local control flow [60] and call-by-value execution [5, 46]. A further development was the advent of algorithmic game semantics, which seeks to use the concrete yet highly accurate nature of game-semantic models to verify programs. Ghica and McCusker showed that observational equivalence of second-order programs of the call-by-name higher-order language with state Idealized Algol (IA) was decidable by representing the game semantics as regular expressions [42]. Further results followed and a complete picture of the decidable fragments of IA emerged [72, 86, 81, 74, 76].

Perhaps surprisingly, the related call-by-value language RML (which can be viewed as a restriction of Standard ML to ground-type references) received comparatively less attention. Prior to the work described in Chapters 3 and 4, there were two papers which described decidable (and undecidable) fragments of RML [40, 73]. However, these decidable fragments were not especially higher-order and did not come close to completing the picture of where the boundary between decidability and undecidability lay.

Two recent papers which do consider decidability in call-by-value stateful languages are [78, 79]. In the first of these, Murawski and Tzevelekos consider a variant of RML where variables can only be allocated in a block-structured manner (as in IA). Removing the ability to pass a reference out of the scope in which it is declared proves to be quite restrictive and the authors are able to show decidability at type sequents which are known to be undecidable in RML. In the second of their papers, they consider Reduced ML. That is, RML without the "bad variable"-constructor. The difference between the equivalence relations of the two languages only shows up in type sequents with negative occurrences of int ref. However, the game-semantic model of Reduced ML is more complex than that of RML, relying on nominal games [77]. Due to this, the fragment that is shown decidable does not contain higher-order functions.

Tools which rely on the decidability results for IA have also been implemented [1] and techniques such as on-the-fly model checking [12], data abstraction [35], predicate abstraction [13] and syntactic approximation [41] have been applied to help improve the performance. However, these tools all check weaker reachability properties rather than equivalence and do not consider programs of order higher than two.

1.4 Thesis Overview

In this thesis we investigate the use of game semantics for the automatic equivalence checking of open higher-order programs. We believe the game-semantic approach has several benefits. Firstly, it has a solid theoretical foundation being based on full abstraction results. If we wish to prove anything about a program we should have some mathematical framework upon which the proof relies. Secondly, the algorithms based on game semantics are fully automatic. They do not require any annotation on the part of the user. The identification of the decidable fragments of languages allows us to know exactly when the methods will work. The algorithms are decision procedures and do not rely on any over- or under-approximation so within the decidable fragment the algorithm will always (eventually) be able to tell whether two given programs are equivalent or not. Further, in the case of inequivalence it is straightforward to construct a counterexample (according to Clarke and Veith "counterexamples are the single most effective feature to convince system engineers about the value of formal verification" [31]). Finally, the game-semantic method allows us to verify *open* programs. That is, we do not need to consider a complete program as a monolithic piece of code, but can break it up and concentrate on verifying the modules or functions of interest on their own. Our programs may contain references to functions whose definitions are unknown.

Our work aims towards getting a complete classification of the decidable fragments of RML, in a similar manner to what was achieved for IA. Further, we implement prototype tools for equivalence-checking higher-order programs to illustrate the power of the game-semantic method and to investigate whether these algorithms can be used in practice.

We start out in Chapter 2 by reviewing the background definitions and results we will rely on in subsequent chapters. In particular, we introduce the languages IA and RML and their operational semantics before presenting their game-semantic models. We also recap previous results in algorithmic game semantics.

In Chapter 3 we look at the O-strict fragment of RML. This is the largest fragment of RML for which, in the game-semantic model, justification pointers from O-moves can always be uniquely reconstructed. This turns out to consist of terms of short types (order at most two, arity at most one) which may contain free identifiers as long as their argument types are also short. This fragment contains large and complex higher-order types and includes many examples from the literature which are known to be challenging to handle. It is strictly larger than the fragments of RML previously identified as having decidable equivalence problems. We first show how to precisely represent the strategy denotations of terms as languages over finite alphabets. This is non-trivial as justification pointers from P-moves need to be encoded and existing methods do not seem sufficient. The main contribution of this chapter is then to show that these languages can be recognised by Visibly Pushdown Automata (VPA) constructed inductively over the structure of terms. This shows that the O-strict fragment of RML has a decidable observational equivalence problem. Further, by reduction of the equivalence problem for nondeterministic automata on binary trees we show that the problem is ExpTime-complete.

Next, in Chapter 4 we look at extensions to the O-strict fragment. We first consider adding recursion to the language. In a similar result to that of [76] we show that recursive first-order arity one functions can be added without affecting decidability, although VPA are no longer sufficiently expressive and Deterministic Pushdown Automata must be used instead. We then consider non-O-strict type sequents. We use deterministic Class Memory Automata, a form of automata over infinite alphabets, to show that observational equivalence is decidable for first-order terms of arity two. Such terms are not O-strict and the infinite alphabet is needed to encode the location of justification pointers from O-moves. Finally, we consider at which point observational equivalence becomes undecidable. From previous results in the literature it is known that third-order programs are undecidable [72, 73]. Further, we adapt existing proofs to show that equivalence is undecidable in the presence of recursive second-order functions or (non-recursive) second-order functions of arity two or more if there is a first-order argument which is not the final argument. These proofs can also be used to show the problem is undecidable if free identifiers whose argument types include one of the undecidable types are present.

Subsequently, in Chapter 5 we consider languages which contain non-local control flow. Game semantically, this is known to correspond to dropping the bracketing condition [60]. We show that in call-by-name IA, adding a **catch**-construct does not affect the decidability of the third-order fragment. The proof relies on the fact that

in the absence of bracketing the visibility condition still implies a form of weak bracketing. Under this condition, non-local jumps of control (i.e. bracketing violations) correspond to popping the control-stack and we can make such pops explicit when we represent the semantics using pushdown automata. Further, we show how to decide the containment relation of [75], proving that decidability holds both with and without the presence of a "bad-variable" constructor in the language.

In the final chapters we move away from theoretical results and consider the implementation of these algorithms. In Chapter 6 we present our model checkers HOMER and HECTOR. HOMER is an equivalence checker for the third-order fragment of IA based on the result of [81]. It takes two terms in β-normal form, constructs VPA which precisely capture their game semantics and checks them for language equivalence. In case the given terms are inequivalent, HOMER constructs both a game-semantic and an operational-semantic counterexample in the form of a play and a separating context respectively. We believe HOMER was the first model checker for third-order programs. Our tool HECTOR is an implementation of the result of Chapter 3 and checks equivalence for terms of O-strict RML. It takes advantage of on-the-fly model checking which allows it to avoid constructing the whole model if it finds a counterexample early. This makes HECTOR particularly effective on inequivalences. We showcase the abilities of both tools on a number of examples and evaluate their performance, where appropriate comparing with the tool MAGE [12].

Finally, in Chapter 7 we investigate the application of symbolic execution to improving the performance of our model checkers. Symbolic execution involves executing a program using symbolic formulas rather than concrete values. We present a representation of finite automata in which transitions are labelled with formulas and guards rather than concrete letters. The aim is to take advantage of the symmetries of the game-semantic model to represent the denotation of terms more compactly and then utilise efficient SMT solvers to model check them. We evaluate our implementation on a number of examples. The symbolic approach gives impressive performance improvements on examples with large data types, but unfortunately is not as fast when smaller data types are considered.

Chapter 2

Preliminaries

In this chapter we review many of the important definitions and results from the literature upon which the subsequent chapters will rely. In particular, we introduce the programming languages Idealized Algol and RML and present their fully abstract game-semantic models. We also present some of the main results in algorithmic game semantics which utilise these fully abstract models to achieve decidability results.

2.1 Idealized Algol

We start by introducing Idealized Algol (IA) [94]. IA is a prototypical programming language which combines both functional and imperative features. It can be thought of as a call-by-name variant of ML. The syntax and typing rules are presented in Figure 2.1. As can be seen, IA is essentially the (call-by-name) simply-typed λ-calculus augmented with imperative constructs. The base types are com of commands, exp of natural number valued expressions and var of natural number valued variables. Of the IA constructs, the only unusual one is the so-called "bad-variable" constructor **mkvar**. This comes from taking an object-oriented view of state, with variables being the combination of a read method and a write method. While we do not include any arithmetic operations other than $\mathbf{succ}(i)$ and $\mathbf{pred}(i)$, this is sufficient to define all the usual operations and comparisons on natural numbers.

The operational semantics is defined for terms of the form $x_1 : \mathsf{var}, \ldots, x_n : \mathsf{var} \vdash M : \theta$, by a big-step relation $s, M \Downarrow s', V$ given in Figure 2.2. Here s and s' are stores, partial functions from $\{ x_1, \ldots, x_n \}$ to the natural numbers and $x_1 : \mathsf{var}, \ldots, x_n : \mathsf{var} \vdash V : \theta$ is a term in canonical form. The canonical forms are defined by

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

$$\frac{}{\Gamma \vdash \mathbf{skip} : \mathsf{com}} \qquad \frac{i \in \mathbb{N}}{\Gamma \vdash i : \mathsf{exp}} \qquad \frac{\Gamma \vdash M : \mathsf{exp}}{\Gamma \vdash \mathbf{succ}(M) : \mathsf{exp}} \qquad \frac{\Gamma \vdash M : \mathsf{exp}}{\Gamma \vdash \mathbf{pred}(M) : \mathsf{exp}}$$

$$\frac{}{\Gamma, x : \theta \vdash x : \theta} \qquad \frac{\Gamma \vdash M : \mathsf{exp} \qquad \Gamma \vdash N_0 : \beta \qquad \Gamma \vdash N_1 : \beta}{\Gamma \vdash \mathbf{if}\, M\, \mathbf{then}\, N_0\, \mathbf{else}\, N_1 : \beta} \qquad \frac{\Gamma \vdash M : \mathsf{com} \qquad \Gamma \vdash N : \beta}{\Gamma \vdash M; N : \beta}$$

$$\frac{\Gamma \vdash M : \mathsf{exp} \qquad \Gamma \vdash N : \mathsf{com}}{\Gamma \vdash \mathbf{while}\, M\, \mathbf{do}\, N : \mathsf{com}} \qquad \frac{\Gamma \vdash M : \mathsf{var} \qquad \Gamma \vdash N : \mathsf{exp}}{\Gamma \vdash M := N : \mathsf{com}} \qquad \frac{\Gamma \vdash M : \mathsf{var}}{\Gamma \vdash !M : \mathsf{exp}}$$

$$\frac{\Gamma, X : \mathsf{var} \vdash M : \beta}{\Gamma \vdash \mathbf{new}\, X\, \mathbf{in}\, M : \beta} \qquad \frac{\Gamma, x : \theta \vdash M : \theta'}{\Gamma \vdash \lambda x^{\theta}.M : \theta \to \theta'} \qquad \frac{\Gamma \vdash M : \theta \to \theta' \qquad \Gamma \vdash N : \theta}{\Gamma \vdash MN : \theta'}$$

$$\frac{\Gamma \vdash M : \mathsf{exp} \qquad \Gamma \vdash N : \mathsf{exp} \to \mathsf{com}}{\Gamma \vdash \mathbf{mkvar}(M, N) : \mathsf{var}}$$

We use θ to denote an arbitrary type, whereas β denotes a base type.

Figure 2.1: Typing rules for Idealized Algol

For closed $\vdash M$, we write $M \Downarrow$ if there exist s', V such that $\emptyset, M \Downarrow s', V$. The operational semantics induces a natural notion of equivalence.

Definition 2.1. We say that $\Gamma \vdash M_1 : \theta$ *contextually approximates* $\Gamma \vdash M_2 : \theta$, written $\Gamma \vdash M_1 \sqsubseteq_{\approx} M_2$, if for all contexts $C[-]$ such that $\Gamma \vdash C[M_1], C[M_2] : \mathsf{com}$, whenever $C[M_1] \Downarrow$ then we also have $C[M_2] \Downarrow$. If $\Gamma \vdash M_1 \sqsubseteq_{\approx} M_2$ and $\Gamma \vdash M_2 \sqsubseteq_{\approx} M_1$ then we say $\Gamma \vdash M_1 : \theta$ and $\Gamma \vdash M_2 : \theta$ are *contextually equivalent* (or *observationally equivalent*), written $\Gamma \vdash M_1 \cong M_2$.

Observational equivalence is a compelling notion of program equivalence. Two programs are equivalent if, in any programming context, one can always replace the other without affecting the outcome of the computation. For example, $\lambda x^{\mathsf{exp}}.x$ is observationally equivalent to $\lambda x^{\mathsf{exp}}.\mathbf{new}\, y\, \mathbf{in}\, y := x; \mathbf{if}\, !y = 0\, \mathbf{then}\, 0\, \mathbf{else}\, !y$ as the local variable y and the conditional cannot be detected from the outside. However, while a natural notion of equivalence, the quantification over all programming contexts makes it notoriously difficult to reason about. The following examples are less obvious.

Example 2.1 (No Snapback [84])**.**

$$p : \mathsf{com} \to \mathsf{com} \vdash \mathbf{new}\, X\, \mathbf{in}\, p(X := 1); \mathbf{if}\, !X = 1\, \mathbf{then}\, \Omega\, \mathbf{else}\, \mathbf{skip} \cong p(\Omega)$$

We write Ω for the (definable) term (of a given type) which immediately diverges without doing anything else. Here p cannot directly observe or modify x, neither can it undo any side effects caused by running its argument. So in both terms, if p ever runs its argument then the computation will diverge.

$$\frac{}{s, V \Downarrow s, V} \qquad \frac{s, M \Downarrow s', i}{s, \mathbf{succ}(M) \Downarrow s', i+1} \qquad \frac{s, M \Downarrow s', i+1}{s, \mathbf{pred}(M) \Downarrow s', i} \qquad \frac{s, M \Downarrow s', 0}{s, \mathbf{pred}(M) \Downarrow s', 0}$$

$$\frac{s, M \Downarrow s', 0 \qquad s', N_1 \Downarrow s'', V}{s, \mathbf{if}\, M\, \mathbf{then}\, N_0\, \mathbf{else}\, N_1 \Downarrow s'', V} \qquad \frac{i > 0 \qquad s, M \Downarrow s', i \qquad s', N_0 \Downarrow s'', V}{s, \mathbf{if}\, M\, \mathbf{then}\, N_0\, \mathbf{else}\, N_1 \Downarrow s'', V}$$

$$\frac{s, M \Downarrow s', \mathbf{skip} \qquad s', N \Downarrow s'', V}{s, M; N \Downarrow s'', V} \qquad \frac{s, M \Downarrow s', 0}{s, \mathbf{while}\, M\, \mathbf{do}\, N \Downarrow s', \mathbf{skip}}$$

$$\frac{i > 0 \qquad s, M \Downarrow s', i \qquad s', N \Downarrow s'', \mathbf{skip} \qquad s'', \mathbf{while}\, M\, \mathbf{do}\, N \Downarrow s''', \mathbf{skip}}{s, \mathbf{while}\, M\, \mathbf{do}\, N \Downarrow s''', \mathbf{skip}}$$

$$\frac{s, N \Downarrow s', i \qquad s', M \Downarrow s'', x}{s, M := N \Downarrow s''[x \mapsto i], \mathbf{skip}} \qquad \frac{s, M \Downarrow s', x}{s, !M \Downarrow s', s'(x)} \qquad \frac{s[X \mapsto 0], M \Downarrow s', V}{s \backslash X, \mathbf{new}\, X\, \mathbf{in}\, M \Downarrow s' \backslash X, V}$$

$$\frac{s, M \Downarrow s', \lambda x^{\theta}.M' \qquad s', M'[N/x] \Downarrow s'', V}{s, M\, N \Downarrow s'', V} \qquad \frac{s, M \Downarrow s', \mathbf{mkvar}(M', N') \qquad s', M' \Downarrow s'', i}{s, !M \Downarrow s'', i}$$

$$\frac{s, N \Downarrow s', i \qquad s', M \Downarrow s'', \mathbf{mkvar}(M', N') \qquad s'', N' i \Downarrow s''', \mathbf{skip}}{s, M := N \Downarrow s''', \mathbf{skip}}$$

Figure 2.2: Operational semantics of Idealized Algol

Example 2.2 (Scope Extrusion)**.** Consider the three terms

$$M_1 \equiv \lambda F^{(\mathsf{exp}\to\mathsf{exp})\to\mathsf{exp}}.\mathbf{new}\, x := 0\, \mathbf{in}\, F\, (\lambda y^{\mathsf{exp}}.\mathbf{if}\, !x = 0\, \mathbf{then}\, x := y\, \mathbf{else}\, x := y - 1;\, !x)$$

$$M_2 \equiv \lambda F^{(\mathsf{exp}\to\mathsf{exp})\to\mathsf{exp}}.F\, (\lambda y^{\mathsf{exp}}.\mathbf{new}\, x := 0\, \mathbf{in}\, \mathbf{if}\, !x = 0\, \mathbf{then}\, x := y\, \mathbf{else}\, x := y - 1;\, !x)$$

$$M_3 \equiv \lambda F^{(\mathsf{exp}\to\mathsf{exp})\to\mathsf{exp}}.F\, (\lambda y^{\mathsf{exp}}.y)$$

The terms M_1 and M_2 differ only on the scope of the $\mathbf{new}\, X\, \mathbf{in}$-block. Now $M_1 \ncong M_2 \cong M_3$. The first two terms are inequivalent because if F evaluates its argument more than once, in M_1 the value of x will be preserved between calls, whereas in M_2 it will be reset to 0 each time. Because of this, F's argument in M_2 is equivalent to $\lambda y^{\mathsf{exp}}.y$ which is why $M_2 \cong M_3$.

It is worth noting that the unusual **mkvar** construct does not affect observational equivalence [67]. Two terms are equivalent if and only if there is no **mkvar**-free context which separates them (although the same result does not hold for contextual approximation).

2.2 Game Semantics

To help us reason about observational equivalence in IA we will use game semantics. Game semantics is a way of giving meaning to a program which views program execution as the playing of a game between the program and its environment. The game-semantic model for IA is presented by Abramsky and McCusker in [6] and we review the basic definitions here.

2.2.1 Games and Plays

Definition 2.2. An *arena* A is given by a tuple $\langle M_A, \lambda_A, \vdash_A \rangle$ where:

- M_A is a set of *moves*.
- $\lambda_A : M_A \to \{\mathsf{O}, \mathsf{P}\} \times \{\mathsf{Q}, \mathsf{A}\}$ is a labelling function which indicates whether each move belongs to Opponent (O) or Player (P) and whether it is a *question* or an *answer*. We sometimes write λ_A^{OP} for the O/P projection of λ_A and similarly with λ_A^{QA}.
- $\vdash_A \subseteq (M_A \uplus \{\star\}) \times M_A$ is an *enabling relation* which satisfies:
  - $\star \vdash_A m \Rightarrow (\lambda_A(m) = \mathsf{OQ} \wedge [n \vdash_A m \Leftrightarrow n = \star])$.
  - $m \vdash_A n \wedge \lambda_A^{\mathsf{QA}}(n) = \mathsf{A} \Rightarrow \lambda_A^{\mathsf{QA}}(m) = \mathsf{Q}$.
  - $m \vdash_A n \wedge m \neq \star \Rightarrow \lambda_A^{OP}(m) \neq \lambda_A^{OP}(n)$.

The idea of the enabling relation is that when playing a game, a move can only be played if another move has previously been played which enables it. Games will start with an *initial move*, which is enabled by $\star$ and must be an O-question. We write I_A for the set of initial moves from arena A. The other conditions state that answers can only be enabled by questions and that a move belonging to one of the players can only be enabled by a move belonging to the other.

In all the arenas we will consider, every question will enable at least one answer (and so it can be answered). In the model for IA it will also be true that (non-initial) questions can only be enabled by other questions, although this condition will not hold when we consider the model of a call-by-value language. Where it aids clarity we will often present arenas in a graphical form, for example the arena in Figure 2.3. Here the topmost move, q_0, is initial (so an O-question) and each move enables its children. This determines the ownership of each move due to the condition that a move can only enable moves of opposite ownership. While this tree-like form does not

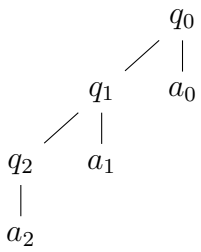


Figure 2.3: Example of a graphically represented arena

contain information about λ_A^{QA}, it should be understood that a move q_i is supposed to be a question and a move a_i is an answer.

To play a game over the arena A, the two players must take it in turns to play a move from M_A which is enabled by a previously played move. This will form a *justified sequence*, a sequence of moves in which each non-initial move m is equipped with a *justification pointer* to a previous move n such that $n \vdash_A m$. We say that n *justifies* m. Similarly, if we can reach n' by following a sequence of justification pointers from m' then we say that n' *hereditarily justifies* m'. Note that the first move in a justified sequence must be an initial move (as there are no possible justifying moves) and so justified sequences always start with O-questions. We often use s, t, u to range over (justified) sequences of moves, m, n to range over moves and q or a to range over question- or answer-moves respectively. A justified sequence s will be called *legal* if the following conditions hold:

- **Alternation**: The players take it in turns to play. If $s = s_1 \cdot m \cdot n \cdot s_2$ then $\lambda^{OP}(m) \neq \lambda^{OP}(n)$.

- **Bracketing**: We say that a question q is *answered* by a in the justified sequence s if q justifies a. The bracketing condition holds if for every prefix $t \cdot q \cdot u \cdot a$ of s with q answered by a, it is the case that every question asked in u has been answered (in u). That is, whenever an answer is played, it must answer the most recently asked unanswered question.

- **Visibility**: If $t \cdot m$ is a prefix of s with m non-initial, then the justifier of m occurs in $\mathsf{view}(s)$. The view function is intended to represent the "currently

relevant" subsequence of moves. It is defined inductively by

$$\begin{aligned} \mathsf{view}(\epsilon) &= \epsilon \\ \mathsf{view}(sm) &= m \qquad \text{if } \star \vdash m \\ \mathsf{view}(\ s\ \widehat{m\ t\ n}\ ) &= \mathsf{view}(s)\ \widehat{m\ n}\ . \end{aligned}$$

We will often use the term O-view to refer to the view of an even-length sequence (i.e. by alternation the O-view is the view when it is O's turn to play). Similarly, by P-view we mean the view of an odd-length sequence.

We will write L_A for the set of legal sequences over the arena A. It turns out to be useful to restrict the set of plays for a game to a subset of L_A. For example, we commonly insist that our games are *well-opened* meaning that each play can contain at most one initial move. To achieve this we define a *game* A to consist of a pair of an arena $\langle M_A, \lambda_A, \vdash_A \rangle$ and a non-empty, prefix-closed set P_A of legal sequences on this arena which must satisfy that if $s \in P_A$ and I is a set of initial moves of s then $s \upharpoonright I \in P_A$. Here we use $s \upharpoonright I$ to denote the subsequence of s consisting of all moves hereditarily justified by an initial move $i \in I$. If $s \in P_A$ we say that s is a *play*.

When representing plays, we sometimes omit the justification pointers and only present the underlying move sequence. We usually only do this in situations when the rules of the game can be used to uniquely determine the positions of the omitted pointers from the underlying move sequence.

2.2.2 Constructions on Games

We will use games to represent IA types. The games for the base types are given in Figure 2.4. We will also have use for the empty game which we denote $\mathsf{1}$. From these basic games we will construct more complex games using the constructions in Figure 2.5. Here we use $s \upharpoonright A$ to mean the subsequence of the legal sequence s consisting of moves from the arena A, $s \upharpoonright m$ to mean the subsequence of the legal sequence s consisting of moves hereditarily justified by the occurrence of move m and $\overline{\lambda_A}$ for the OP-complement of λ_A. The game $A \times B$ consists of the union of A and B; a play of $A \times B$ must be either a play from A or a play from B. In contrast, a play from $A \multimap B$ is an interleaving of a play from A and a play from B. However, the ownership of moves in A is reversed so the play must start in B with initial moves from A being justified by initial moves from B. Finally, plays of $!A$ consist

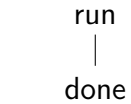
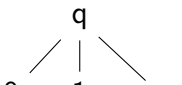
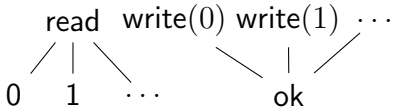
Type	Game	Graphical Representation
com	$M_{[\![\mathsf{com}]\!]} = \{\, \mathsf{run}, \mathsf{done} \,\}$ $\lambda_{[\![\mathsf{com}]\!]} = \mathsf{run} \mapsto \mathsf{OQ}, \mathsf{done} \mapsto \mathsf{PA}$ $\vdash_{[\![\mathsf{com}]\!]} = \{\, (\star, \mathsf{run}), (\mathsf{run}, \mathsf{done}) \,\}$ $P_{[\![\mathsf{com}]\!]} = \{\, \epsilon, \mathsf{run}, \widehat{\mathsf{run}\ \mathsf{done}} \,\}$	
exp	$M_{[\![\mathsf{exp}]\!]} = \{\, \mathsf{q} \,\} \cup \mathbb{N}$ $\lambda_{[\![\mathsf{exp}]\!]} = \mathsf{q} \mapsto \mathsf{OQ}, \mathsf{i} \mapsto \mathsf{PA}$ $\vdash_{[\![\mathsf{exp}]\!]} = \{\, (\star, \mathsf{q}) \,\} \cup \{\, (\mathsf{q}, \mathsf{i}) \mid i \in \mathbb{N} \,\}$ $P_{[\![\mathsf{exp}]\!]} = \{\, \epsilon, \mathsf{q} \,\} \cup \{\, \widehat{\mathsf{q}\ \mathsf{i}} \mid i \in \mathbb{N} \,\}$	
var	$M_{[\![\mathsf{var}]\!]} = \{\, \mathsf{read}, \mathsf{ok} \,\} \cup \mathbb{N} \cup \{\, \mathsf{write}(i) \mid i \in \mathbb{N} \,\}$ $\lambda_{[\![\mathsf{var}]\!]} = \mathsf{read} \mapsto \mathsf{OQ}, \mathsf{write}(i) \mapsto \mathsf{OQ}, \mathsf{ok} \mapsto \mathsf{PA}, \mathsf{i} \mapsto \mathsf{PA}$ $\vdash_{[\![\mathsf{var}]\!]} = \{\, (\star, \mathsf{read}) \,\} \cup \{\, (\star, \mathsf{write}(i)) \mid i \in \mathbb{N} \,\} \cup \{\, (\mathsf{write}(i), \mathsf{ok}) \mid i \in \mathbb{N} \,\} \cup \{\, (\mathsf{read}, \mathsf{i}) \mid i \in \mathbb{N} \,\} \cup$ $P_{[\![\mathsf{var}]\!]} = \{\, \epsilon, \mathsf{read} \,\} \cup \{\, \mathsf{write}(i) \mid i \in \mathbb{N} \,\} \cup \{\, \widehat{\mathsf{read}\ \mathsf{i}} \mid i \in \mathbb{N} \,\} \cup \{\, \widehat{\mathsf{write}(i)\ \mathsf{ok}} \mid i \in \mathbb{N} \,\}$	

Figure 2.4: Games for the IA base types

of interleavings of well-opened plays of A. We use these last two constructions to define $A \Rightarrow B = !A \multimap B$. A play from this game consists of a single play from B interleaved with several plays from A. Although several of these constructions are defined in terms of a simple interleaving of plays, the rules of the games (and in particular alternation) imply that this interleaving must obey certain *switching conditions*. Plays in the game $A \multimap B$ must consist of a play from A interleaved with a play from B, but for this to be legal only P will be able to switch between A and B; O must play in the same component as P's last move. Conversely, play in $!A$ consists of interleavings of plays from A but this time only O can switch between threads. We will be considering games of the form $(A_1 \times \ldots \times A_n) \Rightarrow B$ and in this game only P can switch between components or between A_i-threads.

2.2.3 Strategies

Having defined games and plays we can now define strategies. While games will be the denotation of types, strategies will be the denotation of terms, setting out how the execution of a program will play out over its type.

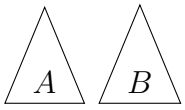
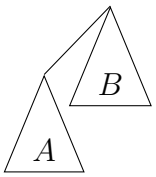
Formal Definition			Graphical Representation
$M_{A\times B}$	$=$	$M_A \uplus M_B$	
$\lambda_{A\times B}$	$=$	$[\lambda_A, \lambda_B]$	
$\star \vdash_{A\times B} n$	$\Leftrightarrow$	$\star \vdash_A n \vee \star \vdash_B n$	
$m \vdash_{A\times B} n$	$\Leftrightarrow$	$m \vdash_A n \vee m \vdash_B n$	
$P_{A\times B}$	$=$	$\{ s \in L_{A\times B} \mid s \upharpoonright A \in P_A \wedge s \upharpoonright B = \epsilon \}$	
	$\cup$	$\{ s \in L_{A\times B} \mid s \upharpoonright B \in P_B \wedge s \upharpoonright A = \epsilon \}$	
$M_{A\multimap B}$	$=$	$M_A \uplus M_B$	
$\lambda_{A\multimap B}$	$=$	$[\overline{\lambda_A}, \lambda_B]$	
$\star \vdash_{A\multimap B} n$	$\Leftrightarrow$	$\star \vdash_B n$	
$m \vdash_{A\multimap B} n$	$\Leftrightarrow$	$m \vdash_A n \vee m \vdash_B n \vee (\star \vdash_B m \wedge \star \vdash_A n)$	
$P_{A\multimap B}$	$=$	$\{ s \in L_{A\multimap B} \mid s \upharpoonright A \in P_A \wedge s \upharpoonright B \in P_B \}$	
$M_{!A}$	$=$	M_A	
$\lambda_{!A}$	$=$	λ_A	
$\vdash_{!A}$	$=$	$\vdash_A$	
$P_{!A}$	$=$	$\{ s \in L_{!A} \mid$ for each initial $m, s \upharpoonright m \in P_A \}$	

Figure 2.5: Basic constructions on games

Definition 2.3. A *strategy* σ for a game A is a non-empty set of even-length plays of A satisfying:

- $smn \in \sigma \Rightarrow s \in \sigma$.
- $smn, smn' \in \sigma \Rightarrow smn = smn'$.

We can think of strategies as being a playbook telling P how to play. The conditions state that σ only contains plays reachable when following strategy σ and that strategies are deterministic. Given an odd length play of A, σ determines at most one move that P should respond with.

We will be considering game-semantic models for languages with state. In such models a strategy's response can depend on the entire history of the play so far. However, we will sometimes find it useful to refer to *innocent* strategies for which this is not the case. Innocent strategies are used to give models of pure functional languages [52]. A strategy is innocent if its response depends only on the view of the play. That is, a strategy $\sigma : A$ is innocent if and only if

$$smn \in \sigma \wedge t \in \sigma \wedge tm \in P_A \wedge \mathsf{view}(tm) = \mathsf{view}(sm) \Rightarrow tmn \in \sigma.$$

2.2.3.1 Composition

Perhaps the most important operation on strategies is composition. Given two strategies $\sigma : A \multimap B$ and $\tau : B \multimap C$ we wish to compose them to form a strategy $\sigma;\tau : A \multimap C$. This will be done by a CSP-style synchronisation plus hiding. To define it formally, we first need some auxiliary definitions.

Let u be a sequence of moves from A, B and C together with justification pointers for all moves except for those initial in C. Define $u \upharpoonright A, B$ to consist of the subsequence of u containing only moves from A and B; if any of these moves had a pointer to a move from C then delete that pointer. We define $u \upharpoonright B, C$ similarly. We say that u is an *interaction sequence* of A, B and C if $u \upharpoonright A, B \in P_{A \multimap B}$ and $u \upharpoonright B, C \in P_{B \multimap C}$. The set of such sequences is denoted $\mathsf{int}(A, B, C)$.

Note that for an interaction sequence u, the only pointers from A- or C-moves to B-moves are from initial A-moves to initial B-moves. Similarly, the only pointers from B-moves to A- or C-moves are from initial B-moves to initial C-moves. This allows us to define $u \upharpoonright A, C$ as the subsequence of u containing only moves from A and C but with any pointers from an initial A-move, i, to an initial B-move, j, redirected to point at the initial C-move, k, which justified j.

We can now define the composition of two strategies $\sigma : A \multimap B$ and $\tau : B \multimap C$.

$$\sigma;\tau = \{\, u \upharpoonright A, C \mid u \in \mathsf{int}(A, B, C) \wedge u \upharpoonright A, B \in \sigma \wedge u \upharpoonright B, C \in \tau \,\}$$

In our model of IA we will need to be able to compose maps $\sigma : !A \multimap B$ and $\tau : !B \multimap C$ (i.e. $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$). To do this we need to be able to promote σ to a map $\sigma^\dagger : !A \multimap !B$.

$$\sigma^\dagger = \{\, s \in L_{!A \multimap !B} \mid \text{for all initial } m, s \upharpoonright m \in \sigma \,\}$$

We can then define composition of $\sigma : !A \multimap B$ and $\tau : !B \multimap C$ by $\sigma \fatsemi \tau = \sigma^\dagger;\tau : !A \multimap C$. The category with objects as well-opened games and morphisms $A \to B$ as strategies for $!A \multimap B$ is well-defined and cartesian closed.

2.2.3.2 Copycat Strategies

An important class of strategies are the copycat strategies. These are strategies in which P always responds to O by repeating O's move but in a different component. For any game A we define $\mathsf{id}_A : A \multimap A$ as below. We use subscripts to differentiate between the copies of A on different sides of the disjoint sum.

$$\mathsf{id}_A = \{\, s \in P_{A_1 \multimap A_2} \mid \text{ for all prefixes } t \text{ of } s \text{ we have } t \upharpoonright A_1 = t \upharpoonright A_2 \,\}$$

These strategies are the identity strategies. That is, for all $\sigma : A \multimap B$, $\mathsf{id}_A; \sigma = \sigma = \sigma; \mathsf{id}_B$.

Similarly, we can define dereliction strategies $\mathsf{der}_A : !A \multimap A$ as copycats.

$$\mathsf{der}_A = \{\, s \in P_{!A \multimap A} \mid \text{ for all prefixes } t \text{ of } s \text{ we have } t \upharpoonright !A = t \upharpoonright A \,\}$$

These dereliction strategies form the identities for the promoted form of composition $\fatsemi$.

2.2.4 Game Semantics of Idealized Algol

In the game-semantic model for IA, types will be interpreted as games. The games for the base types were already presented in Figure 2.4. We will interpret functional types by $[\![\theta_1 \to \theta_2]\!] = [\![\theta_1]\!] \Rightarrow [\![\theta_2]\!]$ and typing contexts by $[\![x_1 : \theta_1, \ldots, x_n : \theta_n]\!] = [\![\theta_1]\!] \times \ldots \times [\![\theta_n]\!]$. The denotation of a term $[\![\Gamma \vdash M : \theta]\!]$ is then a strategy for the game $[\![\Gamma]\!] \Rightarrow [\![\theta]\!]$. These strategies are constructed compositionally over the structure of the term. The definitions are given in Figure 2.6. Free identifiers are interpreted as projections, which are the copycat strategies on $[\![\Gamma]\!] \times [\![\theta]\!] \Rightarrow [\![\theta]\!]$ where P copies O's moves between the two copies of $[\![\theta]\!]$. We interpret λ-abstraction using the currying operator, Λ, which just performs a renaming of moves according to the isomorphism between $[\![\Gamma]\!] \times [\![\theta_1]\!] \Rightarrow [\![\theta_2]\!]$ and $[\![\Gamma]\!] \Rightarrow ([\![\theta_1]\!] \Rightarrow [\![\theta_2]\!])$. Application is performed using pairing (which is simply the disjoint union of strategies) and composition with the ev strategy which is the copycat strategy on $(([\![\theta_1]\!] \Rightarrow [\![\theta_2]\!]) \times [\![\theta_1]\!]) \Rightarrow [\![\theta_2]\!]$ and can be constructed as $\Lambda^{-1}(\mathsf{id}_{[\![\theta_1]\!] \Rightarrow [\![\theta_2]\!]})$ where Λ^{-1} just performs the inverse renaming to the currying operator. The strategies used to interpret the remaining IA constructs are interpreted using special strategies which are given in Figure 2.7 as regular expressions describing their complete plays. Justification pointers have been omitted as they can be uniquely reconstructed and the strategies will contain all even-length prefixes of the plays shown. We use subscripts to indicate which component of the arena a move originates from.

This model of IA turns out to be extremely accurate. We say that a play is *complete* if every question asked in it has been answered. Let $\mathbf{comp}(\sigma)$ denote the set of complete plays from strategy σ. We have the following full abstraction result.

$$
\begin{array}{rcl}
[\![\Gamma, x:\theta \vdash x]\!] & = & \pi_x \\
[\![\Gamma \vdash \lambda x.M]\!] & = & \Lambda([\![\Gamma, x \vdash M]\!]) \\
[\![\Gamma \vdash MN]\!] & = & \langle\, [\![\Gamma \vdash M]\!], [\![\Gamma \vdash N]\!] \,\rangle \fatsemi \mathsf{ev} \\
[\![\Gamma \vdash \mathbf{skip}]\!] & = & \{\, \epsilon,\ \widehat{\mathsf{run}\ \ \mathsf{done}}\ \} \\
[\![\Gamma \vdash i : \mathsf{exp}]\!] & = & \{\, \epsilon,\ \widehat{\mathsf{q}\ \ \mathsf{i}}\ \} \\
[\![\Gamma \vdash \mathbf{succ}(M)]\!] & = & [\![\Gamma \vdash M]\!] \fatsemi \mathsf{succ} \\
[\![\Gamma \vdash \mathbf{pred}(M)]\!] & = & [\![\Gamma \vdash M]\!] \fatsemi \mathsf{pred} \\
[\![\Gamma \vdash \mathbf{if}\, M\, \mathbf{then}\, N_0\, \mathbf{else}\, N_1]\!] & = & \langle\, [\![\Gamma \vdash M]\!], [\![\Gamma \vdash N_0]\!], [\![\Gamma \vdash N_1]\!] \,\rangle \fatsemi \mathsf{cond} \\
[\![\Gamma \vdash M; N]\!] & = & \langle\, [\![\Gamma \vdash M]\!], [\![\Gamma \vdash N]\!] \,\rangle \fatsemi \mathsf{seq} \\
[\![\Gamma \vdash \mathbf{while}\, M\, \mathbf{do}\, N]\!] & = & \langle\, [\![\Gamma \vdash M]\!], [\![\Gamma \vdash N]\!] \,\rangle \fatsemi \mathsf{while} \\
[\![\Gamma \vdash M := N]\!] & = & \langle\, [\![\Gamma \vdash M]\!], [\![\Gamma \vdash N]\!] \,\rangle \fatsemi \mathsf{assign} \\
[\![\Gamma \vdash !M]\!] & = & [\![\Gamma \vdash M]\!] \fatsemi \mathsf{deref} \\
[\![\Gamma \vdash \mathbf{new}\, X\, \mathbf{in}\, M]\!] & = & [\![\Gamma \vdash \lambda X^{\mathsf{var}}.M]\!] \fatsemi \mathsf{cell} \\
[\![\Gamma \vdash \mathbf{mkvar}(M, N)]\!] & = & \langle\, [\![\Gamma \vdash M]\!], [\![\Gamma \vdash N]\!] \,\rangle \fatsemi \mathsf{mkvar}
\end{array}
$$

Figure 2.6: Game semantics of Idealized Algol

$$
\begin{array}{l}
\mathsf{succ} : [\![\mathsf{exp}]\!]_L \Rightarrow [\![\mathsf{exp}]\!]_R = \displaystyle\sum_{n\in\mathbb{N}} \mathsf{q}_R \cdot \mathsf{q}_L \cdot \mathsf{n}_L \cdot (\mathsf{n}+1)_R \\
\mathsf{pred} : [\![\mathsf{exp}]\!]_L \Rightarrow [\![\mathsf{exp}]\!]_R = \left(\displaystyle\sum_{n\in\mathbb{N}} \mathsf{q}_R \cdot \mathsf{q}_L \cdot (\mathsf{n}+1)_L \cdot \mathsf{n}_R\right) + \mathsf{q}_R \cdot \mathsf{q}_L \cdot 0_L \cdot 0_R \\
\mathsf{cond} : ([\![\mathsf{exp}]\!]_1 \times [\![\beta]\!]_2 \times [\![\beta]\!]_3) \Rightarrow [\![\beta]\!]_R = \\
\quad \left(\displaystyle\sum_{m\vdash_{[\![\beta]\!]} n, i\in\mathbb{N}} m_R \cdot \mathsf{q}_1 \cdot (\mathsf{i}+1)_1 \cdot m_2 \cdot n_2 \cdot n_R\right) + \left(\displaystyle\sum_{m\vdash_{[\![\beta]\!]} n} m_R \cdot \mathsf{q}_1 \cdot 0_1 \cdot m_3 \cdot n_3 \cdot n_R\right) \\
\mathsf{seq} : ([\![\mathsf{com}]\!]_1 \times [\![\mathsf{com}]\!]_2) \Rightarrow [\![\mathsf{com}]\!]_R = \mathsf{run}_R \cdot \mathsf{run}_1 \cdot \mathsf{done}_1 \cdot \mathsf{run}_2 \cdot \mathsf{done}_2 \cdot \mathsf{done}_R \\
\mathsf{while} : ([\![\mathsf{exp}]\!]_1 \times [\![\mathsf{com}]\!]_2) \Rightarrow [\![\mathsf{com}]\!]_R = \\
\quad \mathsf{run}_R \cdot \left(\displaystyle\sum_{n\in\mathbb{N}} \mathsf{q}_1 \cdot (\mathsf{n}+1)_1 \cdot \mathsf{run}_2 \cdot \mathsf{done}_2\right)^* \cdot \mathsf{q}_1 \cdot 0_1 \cdot \mathsf{done}_R \\
\mathsf{assign} : ([\![\mathsf{var}]\!]_1 \times [\![\mathsf{exp}]\!]_2) \Rightarrow [\![\mathsf{com}]\!]_R = \displaystyle\sum_{n\in\mathbb{N}} \mathsf{run}_R \cdot \mathsf{q}_2 \cdot \mathsf{n}_2 \cdot \mathsf{write}(n)_1 \cdot \mathsf{ok}_1 \cdot \mathsf{done}_R \\
\mathsf{deref} : [\![\mathsf{var}]\!]_L \Rightarrow [\![\mathsf{exp}]\!]_R = \displaystyle\sum_{n\in\mathbb{N}} \mathsf{q}_R \cdot \mathsf{read}_L \cdot \mathsf{n}_L \cdot \mathsf{n}_R \\
\mathsf{cell} : ([\![\mathsf{var}]\!]_1 \Rightarrow [\![\beta]\!]_2) \Rightarrow [\![\beta]\!]_R = \\
\quad \displaystyle\sum_{m\vdash_{[\![\beta]\!]} n} m_R \cdot m_2 \cdot (\mathsf{read}_1 \cdot 0_1)^* \cdot \left(\sum_{i\in\mathbb{N}} \mathsf{write}(i)_1 \cdot \mathsf{ok}_1 \cdot (\mathsf{read}_1 \cdot \mathsf{i}_1)^*\right)^* \cdot n_2 \cdot n_R \\
\mathsf{mkvar} : ([\![\mathsf{exp}]\!]_1 \times ([\![\mathsf{exp}]\!]_2 \Rightarrow [\![\mathsf{com}]\!]_3)) \Rightarrow [\![\mathsf{var}]\!]_R = \\
\quad \left(\displaystyle\sum_{n\in\mathbb{N}} \mathsf{read}_R \cdot \mathsf{q}_1 \cdot \mathsf{n}_1 \cdot \mathsf{n}_R\right) + \left(\displaystyle\sum_{n\in\mathbb{N}} \mathsf{write}(n)_R \cdot \mathsf{run}_3 \cdot (\mathsf{q}_2 \cdot \mathsf{n}_2)^* \cdot \mathsf{done}_3 \cdot \mathsf{ok}_R\right)
\end{array}
$$

Figure 2.7: Complete plays of the strategies for the IA constructs

$$\frac{}{\Gamma \vdash () : \mathsf{unit}} \qquad \frac{i \in \mathbb{N}}{\Gamma \vdash i : \mathsf{int}} \qquad \frac{\Gamma \vdash M : \mathsf{int}}{\Gamma \vdash \mathbf{succ}(M) : \mathsf{int}} \qquad \frac{\Gamma \vdash M : \mathsf{int}}{\Gamma \vdash \mathbf{pred}(M) : \mathsf{int}}$$

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

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

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

Figure 2.8: Syntax of RML

Theorem 2.1 (Full Abstraction for Idealized Algol [6])**.** *For any two IA terms-in-context* $\Gamma \vdash M, N : \theta$ *we have*

$$\Gamma \vdash M \sqsubseteq\!\!\!\!\sim\, N \Leftrightarrow \mathbf{comp}([\![M]\!]) \subseteq \mathbf{comp}([\![N]\!])$$

and so

$$\Gamma \vdash M \cong N \Leftrightarrow \mathbf{comp}([\![M]\!]) = \mathbf{comp}([\![N]\!]).$$

2.3 RML

A related programming language we are interested in is the call-by-value version of IA, RML. The syntax and typing judgements are essentially the same as IA. However, in order to be more in line with the conventions of ML there are some minor changes, such as using the names unit, int and int ref instead of the base types com, exp and var and so the syntax is presented in Figure 2.8.

The operational semantics of RML is defined in a similar way to that of IA, except that we use call-by-value evaluation rather than call-by-name [91]. While this may seem like a small change, it has some large effects. In particular, local variables no longer have a well-defined block-structured scope. Under call-by-value, variables can be declared in one part of the program and passed into another, which is not possible under call-by-name. To account for this, the stores in our semantics will now be partial functions $s : L \to \mathbb{N}$ where L is a countable set of locations. The semantics is presented in Figure 2.9. We will often write $\mathbf{let}\, x = M\, \mathbf{in}\, N$ as syntactic sugar for $(\lambda x.N)M$. Similarly, we sometimes use $M; N$ as an abbreviation for $(\lambda y.N)M$ where y is a fresh variable.

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

$$\frac{}{s, V \Downarrow s, V} \qquad \frac{s, M \Downarrow s', i}{s, \mathbf{succ}(M) \Downarrow s', i+1} \qquad \frac{s, M \Downarrow s', i+1}{s, \mathbf{pred}(M) \Downarrow s', i} \qquad \frac{s, M \Downarrow s', 0}{s, \mathbf{pred}(M) \Downarrow s', 0}$$

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

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

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

$$\frac{s, M \Downarrow s', \lambda x.M' \qquad s', N \Downarrow s'', V \qquad s'', M'[V/x] \Downarrow s''', V'}{s, MN \Downarrow s''', V'} \qquad \frac{s, M \Downarrow s', 0}{s, \mathbf{while}\ M\ \mathbf{do}\ N \Downarrow s', ()}$$

$$\frac{s, M \Downarrow s', n+1 \qquad s', N \Downarrow s'', () \qquad s'', \mathbf{while}\ M\ \mathbf{do}\ N \Downarrow s''', ()}{s, \mathbf{while}\ M\ \mathbf{do}\ N \Downarrow s''', ()}$$

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

Figure 2.9: Operational semantics of RML

RML is related to Reduced ML [90] the canonical restriction of ML to ground type references. The only substantial difference is the inclusion of the bad-variable constructor in RML[1]. Unfortunately, unlike the call-by-name case, this does affect the observational equivalence relation. For example, $x := !x \cong ()$ holds in Reduced ML but not RML. RML contexts are strictly more powerful and so can separate more terms than Reduced ML contexts [77]. However, these extra contexts only make a difference if the typing $\Gamma \vdash \theta$ contains a negative occurrence of $\mathsf{int\ ref}$ (by a negative occurrence we mean occurring in the left-hand scope of an odd number of $\rightarrow$'s and $\vdash$'s). That is, for Reduced ML terms $\Gamma \vdash M, N : \theta$, if $\Gamma \vdash \theta$ does not contain any negative occurrences of $\mathsf{int\ ref}$ then $\Gamma \vdash M \cong_{\mathrm{RML}} N \Leftrightarrow \Gamma \vdash M \cong_{\text{Reduced ML}} N$ and otherwise $\Gamma \vdash M \cong_{\mathrm{RML}} N \Rightarrow \Gamma \vdash M \cong_{\text{Reduced ML}} N$ [73].

2.3.1 Game Semantics of RML

The two main presentations of game-semantic models for call-by-value languages are by Abramsky and McCusker [5] and Honda and Yoshida [46]. Abramsky and McCusker show how to use a Fam($\mathbf{C}$) construction to construct a bicartesian closed category with a strong monad from a cartesian closed category with a form of weak coproduct. This allows a model of call-by-value computation in the style of Moggi [71] to be constructed from a model of call-by-name computation. Applying this construction to the games model, they construct a category in which objects are families of games and a morphism from $\{ A_i \mid i \in I \}$ to $\{ B_j \mid j \in J \}$ consists of a function $f : I \rightarrow J$ and a family of strategies $\{ \sigma_i : !A_i \multimap B_{f(i)} \mid i \in I \}$. In contrast, Honda and Yoshida define call-by-value games directly and then show their model has a strong monad. Superficially their constructions seem quite different. However, the models are isomorphic and once the constructions have been applied the resulting games are actually very similar. That said, our presentation will more closely resemble the Honda and Yoshida approach. While less general, their constructions are more concrete and so seem more suitable for algorithmic analysis. We review their definitions below.

A *call-by-value arena* (simply referred to as an arena if the call-by-value setting is clear) is defined in the same way as the already introduced call-by-name arenas except that the initial moves are P-answers. Note that in order to have complex arenas this implies that we will have answers which justify questions, a situation which does not happen in call-by-name arenas.

[1]RML also does not include a reference equality test, but in the absence of bad-variables this is definable.

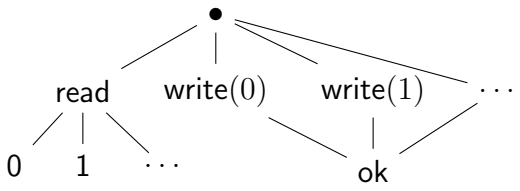


Figure 2.10: Arena for ⟦int ref⟧

We will use call-by-value arenas as the denotation of call-by-value types. The arena ⟦unit⟧ contains a single initial move, •. The arena ⟦int⟧ has the natural numbers as its set of moves, all of which are initial. For ⟦int ref⟧ the situation is more complicated. There is a single initial move, •. This justifies the questions read and write(i) for each $i \in \mathbb{N}$. The moves write(i) have a single answer, ok, whereas the set of answers to read is the natural numbers. This is shown graphically in Figure 2.10.

The definitions of some useful constructions on call-by-value arenas are given in Figure 2.11. Here we use $\overline{I_A}$ as an abbreviation for $M_A \backslash I_A$. Intuitively $A \otimes B$ is the union of the arenas A and B, but with the initial moves combined pairwise. $A \Rightarrow B$ is slightly more complex. First we add a new initial move, •. We take the O/P-complement of A, change the initial moves into questions and set them to now be justified by •. Finally, we take B and set its initial moves to be justified by A's initial moves. Figure 2.12 gives a graphical representation of these constructions using superscripts to denote the ownership of moves and whether they are questions or answers. Note that although the arenas are shown having only a single initial move, this is not guaranteed to be the case. As the depth of the arenas and the (maximal) paths of the enabling relation will add to the complexity of our proofs much more than the breadth of the arenas, when presenting representations of arenas graphically we will often show only a single move when there could in fact be many moves at the same level in the arena.

Remark. Honda and Yoshida actually define arenas to consist of forests [46]. Due to this, their construction for $A \Rightarrow B$ requires distinct copies of B for each initial move of A. If we consider only finite base types, this leads to the number of moves in the denotation of a type being exponential in its size. Abramsky and McCusker's constructions lead to the same blow up [5]. However, we have used a common optimisation [62, 77, 80] in which arenas do not have to be forests and so moves can have multiple enablers. This greater sharing means the number of moves in the denotation of a type grows linearly with its size.

$$
\begin{array}{rcl}
M_{A\Rightarrow B} & = & \{\bullet\} \uplus M_A \uplus M_B \\
\lambda_{A\Rightarrow B} & = & m \mapsto \begin{cases} PA & \text{if } m = \bullet \\ OQ & \text{if } m \in I_A \\ \overline{\lambda_A}(m) & \text{if } m \in \overline{I_A} \\ \lambda_B(m) & \text{if } m \in M_B \end{cases} \\
\star \vdash_{A\Rightarrow B} m & \Leftrightarrow & m = \bullet \\
m \vdash_{A\Rightarrow B} n & \Leftrightarrow & m \vdash_A n \vee m \vdash_B n \vee (m = \bullet \wedge \star \vdash_A n) \vee (\star \vdash_A m \wedge \star \vdash_B n)
\end{array}
$$

$$
\begin{array}{rcl}
M_{A\otimes B} & = & I_A \times I_B \uplus \overline{I_A} \uplus \overline{I_B} \\
\lambda_{A\otimes B} & = & m \mapsto \begin{cases} PA \text{ if } m \in I_A \times I_B \\ \lambda_A(m) \text{ if } m \in \overline{I_A} \\ \lambda_B(m) \text{ if } m \in \overline{I_B} \end{cases} \\
\star \vdash_{A\otimes B} m & \Leftrightarrow & m \in I_A \times I_B \\
m \vdash_{A\otimes B} n & \Leftrightarrow & m \vdash_A n \vee m \vdash_B n \vee \ (m = (i,j) \in I_A \times I_B \wedge (i \vdash_A n \vee j \vdash_B n))
\end{array}
$$

$$
\begin{array}{rcl}
M_{A\to B} & = & M_A \uplus M_B \\
\lambda_{A\to B}(m) & = & \begin{cases} OQ & \text{if } m \in I_A \\ \overline{\lambda_A}(m) & \text{if } m \in \overline{I_A} \\ \lambda_B(m) & \text{if } m \in M_B \end{cases} \\
\star \vdash_{A\to B} m & \Leftrightarrow & \star \vdash_A m \\
m \vdash_{A\to B} n & \Leftrightarrow & m \vdash_A n \vee m \vdash_B n \vee (\star \vdash_A m \wedge \star \vdash_B n)
\end{array}
$$

Figure 2.11: Constructions on call-by-value arenas

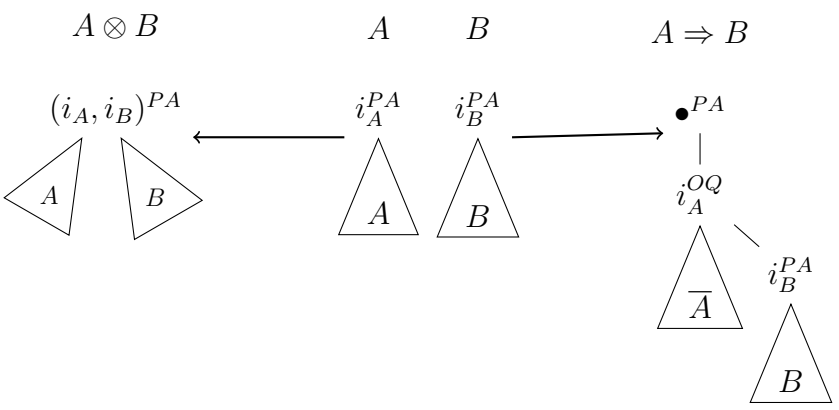


Figure 2.12: Graphical representation of call-by-value constructions

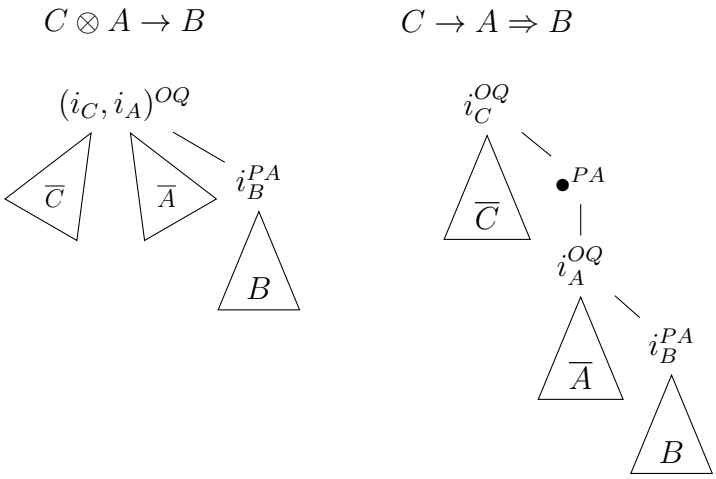


Figure 2.13: Prearenas involved in $p\lambda(\sigma)$

Although we use call-by-value arenas to represent types, we will not actually play games on them. Instead we will play games on *prearenas* in which the initial move is an O-question (that is they are identical in definition to call-by-name arenas). The distinction in definition between the arenas used to represent a type and the prearenas for which the strategies denoting terms will be defined comes about from the monadic nature of the semantics [71]. We can think of it as being the difference between modelling a value of type θ and modelling a computation of type θ (which may either not terminate or have some side-effects). A term $x_1 : \theta_1, \ldots, x_n : \theta_n \vdash M : \theta$ will be represented by a *strategy* for the prearena $[\![\theta_1]\!] \otimes \ldots \otimes [\![\theta_n]\!] \to [\![\theta]\!]$. The construction $\to$ is also given in Figure 2.11 and takes two arenas and produces a prearena. It is essentially the same as $\Rightarrow$ except we omit the (previously initial) move $\bullet$.

In call-by-value game semantics we do not distinguish between a prearena and a game. The set of plays for a prearena is the set of well-opened legal sequences. We also have no need for the promoted form of composition. This is because in function application in call-by-value the argument is always evaluated exactly once, whereas in call-by-name an argument can be evaluated zero or more times. The (unpromoted) composition of strategies $\sigma : A \to B$ and $\tau : B \to C$ is defined in the same manner as the call-by-name case using synchronisation plus hiding.

As in the call-by-name semantics, RML free identifiers will be interpreted by copycat projection strategies. The construction for λ-abstraction is slightly more complex. Given a strategy $\sigma : C \otimes A \to B$, we can construct a strategy $p\lambda(\sigma) : C \to A \Rightarrow B$. The prearenas for these strategies are shown in Figure 2.13. Upon receiving the initial move, say i_C, $p\lambda(\sigma)$ will immediately play $\bullet$. Now O must respond with an

initial A-move, say i_A. Having been given both an initial C-move and an initial A-move P now plays as σ would if given the initial move (i_C, i_A). However, in $C \otimes A \to B$, O can only ever play an initial $C \otimes A$-move once, as it will be the initial move for the prearena. In $C \to A \Rightarrow B$ though, the move i_A is no longer initial and so O may be able to play initial A-moves multiple times. Every time O does play an initial A-move, say i'_A, it will "open a new σ-thread", in which $p\lambda(\sigma)$ will respond as σ would when the initial move is (i_C, i'_A). That is, $p\lambda(\sigma)$ will contain interleavings of plays from σ.

The denotation of an application is defined by

$$[\![\Gamma \vdash MN : \theta_2]\!] = \langle\!\langle\, [\![\Gamma \vdash M : \theta_1 \to \theta_2]\!], [\![\Gamma \vdash N : \theta_1]\!] \,\rangle\!\rangle\, ; \mathsf{ev}.$$

The strategy $\mathsf{ev} : (([\![\theta_1]\!] \Rightarrow [\![\theta_2]\!]) \otimes [\![\theta_1]\!]) \to [\![\theta_2]\!]$ is the natural copycat strategy. The operation $\langle\!\langle\, \cdot, \cdot \,\rangle\!\rangle$ is (partial left) pairing. Given strategies $\sigma : C \to A$ and $\tau : C \to B$, $\langle\!\langle\, \sigma, \tau \,\rangle\!\rangle$ is a strategy on the prearena $C \to A \otimes B$. Play begins with O playing an initial C-move, say i_C. P will then play as σ up to the point at which σ wishes to play an initial A-move, say i_A. At this point, $\langle\!\langle\, \sigma, \tau \,\rangle\!\rangle$ instead starts to play as τ would have played after the initial i_C. This then continues until τ wishes to play an initial B-move, say i_B. Now $\langle\!\langle\, \sigma, \tau \,\rangle\!\rangle$ has both an initial A-move and an initial B-move, so it combines them and plays (i_A, i_B). From this point onwards, P plays as the disjoint union of σ and τ.

The result of these definitions is that $[\![\Gamma \vdash MN]\!]$ will first play as $[\![\Gamma \vdash M]\!]$ up to the point where it is ready to play the unique initial $[\![\theta_1 \to \theta_2]\!]$ move $\bullet$. Then $[\![\Gamma \vdash N]\!]$ takes over, playing up to the point where it is ready to play an initial $[\![\theta_1]\!]$-move. From this point on the two strategies play in parallel, synchronising on moves in $[\![\theta_1]\!]$ which are then hidden.

For conditionals, $[\![\Gamma \vdash \mathbf{if}\, N\, \mathbf{then}\, M_1\, \mathbf{else}\, M_2]\!]$ first runs as $[\![\Gamma \vdash N : \mathsf{int}]\!]$ up to the point where it is ready to play an initial $[\![\mathsf{int}]\!]$-move. If this move is 0 then from then on it plays as $[\![M_2]\!]$, otherwise it plays as $[\![M_1]\!]$. The semantics of $\mathbf{while}\, M\, \mathbf{do}\, N$ is defined similarly.

The remaining RML constructs can be viewed as constants and their strategies are given as regular expressions in Figure 2.14. Just as with IA we have a full abstraction theorem for RML.

Theorem 2.2 (Full Abstraction for RML [5])**.** *For all RML-terms $\Gamma \vdash M, N : \theta$, we have $M \sqsubseteq N$ if and only if $\mathbf{comp}([\![M]\!]) \subseteq \mathbf{comp}([\![N]\!])$.*

Note that unlike in the call-by-name case, a complete play is not necessarily maximal.

$$
\begin{array}{ll}
[\![\Gamma \vdash () : \mathsf{unit}]\!] & i_\Gamma \bullet \\
[\![\Gamma \vdash j : \mathsf{int}]\!] & i_\Gamma\, j \\
[\![\Gamma \vdash \mathsf{succ} : \mathsf{int}_L \to \mathsf{int}_R]\!] & i_\Gamma \bullet \sum_{j\in\mathbb{N}} j_L\,(j+1)_R \\
[\![\Gamma \vdash \mathsf{pred} : \mathsf{int}_L \to \mathsf{int}_R]\!] & i_\Gamma \bullet \left(\sum_{j\in\mathbb{N}} (j+1)_L\, j_R + 0_L\, 0_R\right) \\
[\![\Gamma \vdash \mathsf{deref} : \mathsf{int\ ref}_L \to \mathsf{int}_R]\!] & i_\Gamma \bullet \bullet_L\, \mathsf{read}_L \sum_{j\in\mathbb{N}} j_L\, j_R \\
[\![\Gamma \vdash \mathsf{assign} : \mathsf{int\ ref}_L \to_1 \mathsf{int}_R \to_2 \mathsf{unit}]\!] & i_\Gamma \bullet_1 \bullet_L \bullet_2 \sum_{j\in\mathbb{N}} j_R\, \mathsf{write}(j)_L\, \mathsf{ok}_L \bullet \\
[\![\Gamma \vdash \mathsf{ref} : \mathsf{int\ ref}]\!] & i_\Gamma \bullet (\mathsf{read}\,0)^* \left(\sum_{j\in\mathbb{N}} \mathsf{write}(j)\,\mathsf{ok}\ (\mathsf{read}\,j)^*\right)^*
\end{array}
$$

$$
[\![\Gamma \vdash \mathsf{mkvar} : (\mathsf{unit}_1 \to_2 \mathsf{int}_3) \to_4 (\mathsf{int}_5 \to_6 \mathsf{unit}_7) \to_8 \mathsf{int\ ref}_9]\!]
$$
$$
i_\Gamma \bullet_4 \bullet_2 \bullet_8 \bullet_6 \bullet_9 \left(\mathsf{read}_9 \bullet_1 \sum_{j\in\mathbb{N}} j_3 j_9 + \sum_{k\in\mathbb{N}} \mathsf{write}(k)_9\, k_5 \bullet_7 \mathsf{ok}_9\right)
$$

Justification pointers are uniquely reconstructible and so have been omitted. Subscripts are used to show which component a move originates from. With the exception of ref the strategies will be the minimal innocent strategies containing these plays. The strategy for ref is not innocent and will contain all even-length prefixes of the given plays.

Figure 2.14: Strategies for RML constants

2.4 Algorithmic Game Semantics

We have seen that game semantics can provide highly accurate models of programming languages. These models are also very concrete which lends itself to algorithmic analysis. We intend to use the game-semantic model to automatically decide observational equivalence of higher-order programs. Unfortunately, as both IA and RML are Turing complete, this is in general an undecidable problem. To help overcome this, we will from now on assume that all base types are finite. That is, instead of the natural numbers we use some finite prefix and so exp and int refer to expressions with values in the set $\{0 \ldots n\}$ for some number n (and similarly var and int ref are variables with values in this set). The game-semantic denotations are similarly curtailed (this does not affect the full abstraction results). Unfortunately, this is still not enough for the problem to be decidable. It may seem surprising that equivalence could be undecidable when all base-types are finite, but higher-order functions still introduce infinite behaviours which lead to undecidability [63]. Hence, further restrictions on programs are necessary to achieve decidability. A natural restriction is to allow only programs up to a certain order. Order is a measure of "how higher-order" a program is. We define[2] $\mathbf{ord}(\mathsf{com}) = \mathbf{ord}(\mathsf{exp}) = \mathbf{ord}(\mathsf{var}) = 0$ and $\mathbf{ord}(\theta_1 \to \theta_2) = \mathbf{max}(\mathbf{ord}(\theta_1) + 1, \mathbf{ord}(\theta_2))$. A term $\Gamma \vdash M : \theta$ is then order n if

[2] In RML we define $\mathbf{ord}(\mathsf{int\ ref}) = 1$.

	pure	+**while**	+$\mathbf{Y_0}$	+$\mathbf{Y_1}$
1st-order	coNP	Pspace	DPDA	(=1st-order + $\mathbf{Y_0}$)
2nd-order	Pspace	Pspace	DPDA	Undecidable
3rd-order	Exptime	Exptime	DPDA	Undecidable
4th-order	Undecidable	Undecidable	Undecidable	Undecidable

Table 2.1: Complexity of observational equivalence for fragments of IA

$\mathbf{ord}(\theta) \leq n$ and for every type $x : \theta' \in \Gamma$, $\mathbf{ord}(\theta') < n$. In call-by-name arenas we also refer to the order of a move. Initial moves are order 0, answers have the same order as the question they answer and non-initial questions have order one higher than their justifier. The order of an arena is the maximum order of a move occurring in it. In call-by-name game semantics, order n type sequents give rise to order n arenas.

By restricting the order of programs (and considering only finite base types) decidability results can be achieved.

2.4.1 Call-by-Name

The first steps in using game semantics to automatically decide observational equivalence were taken by Ghica and McCusker [42]. They showed that in plays from the games denoting second-order IA types the location of the justification pointers can always be uniquely reconstructed from the underlying move sequences. This allows us to unambiguously represent a strategy as a set of sequences of moves. Ghica and McCusker went on to show that for the strategies denoting second-order IA terms such sets can be described using regular expressions. This gives a decision procedure for deciding observational equivalence of IA terms.

Further results in this area led to a complete classification of the decidable fragments of IA [72, 86, 81, 74, 76]. These results are summarised in Table 2.1. The rows show the maximum order of allowed programs whereas the columns show what level of recursion is allowed — none at all, allowing loops or including a fixpoint combinator $\mathbf{Y}_i$ for terms of order at most i. The entries labelled DPDA are at least as hard as the DPDA equivalence problem, which is decidable [99] but the only known upper-bound is primitive recursive [104].

Of particular interest is the entry for third-order IA programs with loops [81]. This fragment of IA is referred to as IA^*_3. Plays from games of third-order types no longer have the property that the justification pointers can be uniquely reconstructed

from the underlying move sequence. However, it is only pointers from P-moves which can be ambiguous. These can be encoded by adding tags to the appropriate moves. A second issue is that the sets of plays are not regular. To achieve the ExpTime-bound a class of automata called Visibly Pushdown Automata were used.

2.4.2 Visibly Pushdown Automata

Visibly Pushdown Automata (VPA) are a subclass of pushdown automata in which the stack action is uniquely determined by the input letter [11]. That is, a visibly pushdown alphabet is a tuple $\widetilde{\Sigma} = \langle \Sigma_{push}, \Sigma_{pop}, \Sigma_{noop} \rangle$ partitioning the alphabet into three disjoint finite sets of push-letters, pop-letters and noop-letters and depending on the input letter only the appropriate stack action can be taken. More formally we can define VPA as follows.

Definition 2.4. A (nondeterministic) *Visibly Pushdown Automaton* over the alphabet $\langle \Sigma_{push}, \Sigma_{pop}, \Sigma_{noop} \rangle$ is a tuple $\mathcal{A} = \langle Q, q_0, \Gamma, \delta, F \rangle$ where:

- Q is a finite set of states.
- $q_0 \in Q$ is the initial state.
- Γ is the finite stack alphabet containing a special bottom-of-stack symbol $\perp$.
- $\delta \subseteq (Q \times \Sigma_{push} \times Q \times (\Gamma \backslash \{\perp\})) \cup (Q \times \Sigma_{pop} \times \Gamma \times Q) \cup (Q \times \Sigma_{noop} \times Q)$ is the transition relation.
- $F \subseteq Q$ is the set of final states.

We write Σ for $\Sigma_{push} \cup \Sigma_{pop} \cup \Sigma_{noop}$. For a word $w = a_1 \dots a_k$ in Σ^*, a run of $\mathcal{A}$ on w is a sequence $(q_0, \sigma_0), \dots, (q_k, \sigma_k)$, where each $q_i \in Q$ (and q_0 is the initial state), each σ_i is a stack (a finite sequence from $(\Gamma \backslash \{\perp\})^* \cdot \{\perp\}$), $\sigma_0 = \perp$ and for all $1 \leq i \leq k$ the following hold:

- If $a_i \in \Sigma_{push}$ then for some $\gamma \in \Gamma$, $(q_i, a_i, q_{i+1}, \gamma) \in \delta$ and $\sigma_{i+1} = \gamma \cdot \sigma_i$.
- If $a_i \in \Sigma_{pop}$ then for some $\gamma \in \Gamma$, $(q_i, a_i, \gamma, q_{i+1}) \in \delta$ and either $\gamma \neq \perp$ and $\sigma_i = \gamma \cdot \sigma_{i+1}$ or $\gamma = \perp$ and $\sigma_i = \sigma_{i+1} = \perp$.
- If $a_i \in \Sigma_{noop}$ then $(q_i, a_i, q_{i+1}) \in \delta$ and $\sigma_{i+1} = \sigma_i$.

A run is accepting if the final state $q_k \in F$. The language accepted by $\mathcal{A}$ is the set $\mathcal{L}(\mathcal{A})$ of words w such that $\mathcal{A}$ has an accepting run on w.

VPA are designed for recognising well-bracketed words such as well-formed XML. They can then push when reading an opening bracket and pop when reading a closing bracket. They have extremely desirable closure properties as, unlike for general pushdown automata, it is possible to simulate running two VPA (on the same partitioned input alphabet) in parallel using a product construction as the stacks of the two automata will always be the same height. In particular, for deterministic VPA language equivalence is decidable in polynomial time.

When describing (visibly) pushdown automata we will use the notation $s \xrightarrow{a/\gamma} t$ to mean that the VPA can move from state s to state t, reading input symbol a and pushing stack symbol γ. Similarly, $s \xrightarrow{a,\gamma} t$ denotes moving from s to t reading a while popping γ off the stack.

We will deal mainly with deterministic VPA but will allow our automata to contain internal ϵ-transitions which do not consume any input or use the stack, as long as they are the only outgoing transition from that state. This is acceptable as we can easily "compress out" such transitions (e.g. if $s_1 \xrightarrow{a} s_2 \xrightarrow{\epsilon} s_3$ we can redirect the first transition as $s_1 \xrightarrow{a} s_3$ so we can safely remove the second transition).

2.4.3 Call-by-Value

While the decidable fragments of IA are well understood and naturally partition in line with program order, in RML the situation is much less clear. Prior to the work described in Chapters 3 and 4, there were only two papers in this area [40, 73]. In the second of these Murawski showed that observational equivalence is decidable for terms of the shape $x_1 : \mathit{ctype}_1, \ldots, x_n : \mathit{ctype}_n \vdash M : \mathit{ttype}$ where *ctype* and *ttype* are as follows:

$$\begin{array}{rcl} \mathit{ctype} & ::= & \alpha \mid \alpha \to \beta \\ \alpha & ::= & \beta \mid \beta \to \beta \mid \mathsf{int\ ref} \end{array} \qquad \begin{array}{rcl} \mathit{ttype} & ::= & \alpha \mid \alpha \to \alpha \\ \beta & ::= & \mathsf{unit} \mid \mathsf{int} \end{array}$$

The game semantics for terms of this fragment can be represented using regular expressions. Murawski went on to show that the problem is already undecidable at the type $(\mathsf{unit} \to \mathsf{unit}) \to (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$. This is a second-order type. In contrast, in the call-by-name case the problem does not become undecidable until fourth order.

In the next chapter we extend the known decidable fragment and identify what we believe to be the largest fragment of RML whose game semantics can be precisely captured using VPA.

Chapter 3

O-Strict RML

As we have seen, the fragments of the call-by-name language IA for which observational equivalence is decidable are now well understood. Using the game-semantic model it was shown that the complexity of the problem increases in line with the type theoretic order of the programs considered. Further, to represent the game semantics using automata, the most complex type sequents which are still decidable require the power of pushdown automata. By contrast, under call-by-value the situation appears much less clean. It is known that observational equivalence is already undecidable at second-order, but the largest fragments previously shown to be decidable only required regular expressions to represent. In this chapter we present a candidate for the largest fragment of RML whose semantics is representable by VPA (and so observational equivalence is decidable for this fragment). This work was presented in [47] and is joint work with Andrzej Murawski and Luke Ong.

We start out by identifying the O-strict fragment of RML. This is the largest fragment for which, in the game-semantic representation, the location of justification pointers from O-moves is always uniquely determined. We then discuss how to represent P-pointers by adding tags to moves. This allows us to represent strategies as sets of sequences of letters from a finite alphabet. We then show how to recognise such sets using VPA. This is sufficient to prove observational equivalence is decidable for the O-strict fragment. Finally, we examine the complexity of the problem and show that (when considering terms in canonical form) it is ExpTime-complete.

3.1 The O-Strict Fragment of RML

In order to represent strategies using automata, we need to be able to encode plays (move sequences with pointers) as words. In some cases pointers can be uniquely

reconstructed from the underlying move sequence, thanks to the visibility or well-bracketing conditions which constrain the position of the justifying move. For instance, the targets of pointers from answers can always be deduced from the underlying sequence of moves. In [73] Murawski identified the type sequents of RML for which the corresponding prearenas have the property that their plays can always have their justification pointers uniquely reconstructed in this way. We call this the bi-strict fragment of RML (bi-strict as justification pointers from *both* players are uniquely determined).

Since strategies from the bi-strict fragment can be precisely represented as sets of move sequences, they lend themselves for recognition by automata. Murawski showed that a subset of the bi-strict fragment could be represented using regular expressions. However, we can do better than this. In the call-by-name case it is known that the third-order fragment of IA is decidable, despite not being bi-strict [81]. This result requires tagging moves to encode the location of pointers. However, it is only the location of pointers from P-moves which can be ambiguous in this fragment. This simplifies the problem as the location of P-pointers is determined by the term and so while there may be several legal locations for a pointer, only one can ever occur in a particular strategy. In contrast, pointers from O-moves are much harder to handle as they are controlled by the context. Since in observational equivalence we quantify over all possible contexts, a strategy may include plays with all legal O-pointer locations. In fact, in IA as soon as we move into a fragment where O-pointers are no longer uniquely determined, observational equivalence becomes undecidable.

Guided by the intuition that pointers from P-moves may be encodable by adding tags to moves, yet pointers from O-moves are difficult to handle, we will consider the largest fragment of RML for which, in the game-semantic denotation of terms, O-pointers are always uniquely determined. We refer to this as the *O-strict* fragment of RML ($\mathsf{RML}_{\text{O-Str}}$) and will similarly refer to O-strict prearenas or type sequents.

3.1.1 Types on the Right of O-Strict Sequents

Consider the type sequent $\Gamma \vdash \theta_1 \to \theta_2 \to \theta_3$. Any type sequent in which the type on the right-hand side has arity at least two has this form[1]. The corresponding prearena is shown in Figure 3.1. Notice this has the enabling chain $q_0 \vdash a_0 \vdash q_1 \vdash a_1 \vdash q_2$ (for brevity, we shall say that the arena has a $qaqaq$-branch). The plays $q_0\ a_0\ q_1\ a_1\ q_1\ a_1\ q_2$ and $q_0\ a_0\ q_1\ a_1\ q_1\ a_1\ q_2$ are both valid plays. The only

[1] Arity is the number of arguments a term takes: $\mathbf{arity}(\mathsf{unit}) = \mathbf{arity}(\mathsf{exp}) = 0$, $\mathbf{arity}(\mathsf{int\ ref}) = 1$ and $\mathbf{arity}(\theta_1 \to \theta_2) = \mathbf{arity}(\theta_2) + 1$.

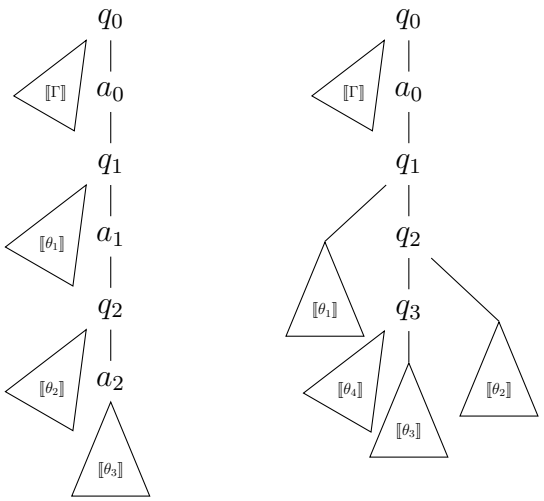


Figure 3.1: Prearenas for $[\![\Gamma \vdash \theta_1 \to \theta_2 \to \theta_3]\!]$ and $[\![\Gamma \vdash ((\theta_4 \to \theta_3) \to \theta_2) \to \theta_1]\!]$

difference between them is the location of the justification pointer from the O-move q_2. Hence, this play is not O-strict and so the O-strict fragment cannot contain any types of arity two or greater.

Another troublesome sequent is $[\![\Gamma \vdash ((\theta_4 \to \theta_3) \to \theta_2) \to \theta_1]\!]$ (a type of order at least 3) for which the prearena is also shown in Figure 3.1. Notice the prearena has a $qaqqq$-branch. In a similar manner, $q_0\ a_0\ q_1\ q_2\ q_1\ q_2\ q_3$ and $q_0\ a_0\ q_1\ q_2\ q_1\ q_2\ q_3$ are two plays over this prearena which differ only in the location of the final O-pointer. Hence, the O-strict fragment cannot contain any types of order three or greater. It should be noted that in RML we consider $\mathsf{int\ ref}$ to have order 1 (and arity 1), in contrast to IA where $\mathbf{ord}(\mathsf{var}) = 0$. We still have that unit and int are order 0.

We have seen that on the right-hand side of a type sequent, we cannot have types of arity two or more, or order three or above. This leaves us with types of order at most two and arity at most one. We will refer to such types as *short*. It turns out that all short-types do result in O-strict prearenas.

Proposition 3.1. *If θ is a short-type then justification pointers from O-moves in $[\![\theta]\!]$ are always uniquely determined by the underlying move sequence in plays over the prearena $[\![\Gamma \vdash \theta]\!]$ (for any Γ).*

Proof. The prearena for $\Gamma \vdash (\beta \to \ldots \to \beta \to \beta) \to \beta$ is shown in Figure 3.2. The prearena for any $\Gamma \vdash \theta$ where θ is short will have this form, although not all moves may be present. Note that the only O-questions are the initial move (which does not have a justification pointer) and q_0 which is justified by the unique occurrence of $\bullet$

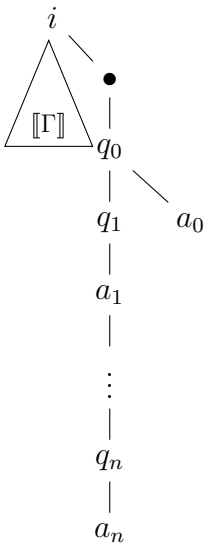


Figure 3.2: Prearena for $\Gamma \vdash (\beta \to \ldots \to \beta \to \beta) \to \beta$

(which answers the initial move and so can occur at most once in a play). The rest of O's moves are answers and so the bracketing condition uniquely determines their justifier. Hence, there can be no ambiguity over the location of O-pointers so all short types can be included in the O-strict fragment. □

3.1.2 Types on the Left of O-Strict Sequents

If θ is a type which gives rise to a non-O-strict prearena when placed on the right-hand side of the turnstile, then we cannot allow θ to occur negatively in any type on the left-hand side (that is positively for the sequent as a whole). This is because the same subset of the prearena which led to the non-O-strict play would occur with the same polarity and so we could again construct a non-O-strict play. Hence, we cannot allow any type which takes an argument of arity two or greater, or order three or greater. So as possible types on the left of the turnstile we are left with types whose arguments are short. We can again check that these are indeed O-strict.

Proposition 3.2. *If Γ is a type environment in which the argument types of all functional free identifiers are short, then in plays over the prearena $[\![\Gamma \vdash \theta_0]\!]$ (for any θ_0) the O-view contains exactly one potential justifier for each legal O-move from $[\![\Gamma]\!]$.*

Proof. The basic shape of the prearenas is shown in Figure 3.3, where each $[\![\theta_i]\!]$ is the arena of a short type, similar to that of Figure 3.2. The only O-questions (on the left-hand side) will be justified by q_i, $i \geq 1$. We claim that there will be at most one

occurrence of each such q_i in the O-view of any (odd-length) play from this prearena. We can show this by induction on i. The base case $i = 1$ is straightforward. As q_1 is justified by the initial move, if it occurs in the O-view it will always be the second move in the view (when constructing the view, if we see a q_1 we would follow its justification pointer back to the initial move, ignoring all moves in between). For the inductive case, consider an odd-length play s with q_{i+1} in the view. The view must have the form $t_1 \overset{\frown}{q_i}\ t_2 \overset{\frown}{a_i}\ \overset{\frown}{q_{i+1}}\ t_3$ where the occurrence of q_{i+1} shown is the right-most and each t_i is some (possibly empty) sequence of moves. Note that we know q_i must also appear in the view since the view of a play is always a play and so a_i's justifier must be present. We know q_{i+1} does not occur in t_3 by construction. Further, $t_1 \cdot q_i = \mathsf{view}(s' \cdot q_i)$ for some prefix $s' \cdot q_i$ of s. By the induction hypothesis this implies that q_i does not occur in t_1. Hence, there can be no a_i in t_1 and so there can be no q_{i+1} either. This only leaves t_2 to check. Suppose there is an occurrence of q_{i+1} in t_2. Then this must be justified by an a_i which itself is justified by a q_i both of which must be visible and so occur in $t_1 \cdot q_i \cdot t_2$. However, $t_1 \cdot q_i \cdot t_2 = \mathsf{view}(s'')$ for some prefix s'' of s. By the induction hypothesis, there can only be one occurrence of q_i in $t_1 \cdot q_i \cdot t_2$ (i.e. q_i does not occur in either t_1 or t_2). Hence, the a_i in $t_1 \cdot q_i \cdot t_2$ must answer this q_i. However, this means we have two moves a_i answering the same occurrence of q_i in s. This contradicts bracketing. Therefore, there cannot be an occurrence of q_{i+1} in t_2. Hence, by induction, the O-view of a play in this arena can contain at most one occurrence of each q_i. So, the O-view contains at most one potential justifier for each O-move. Hence, this prearena is O-strict. □

We have now shown that the O-strict fragment of RML consists of terms of short type and for which the arguments of any free identifiers are also short. That is, we can define $\mathsf{RML}_{\text{O-Str}}$ as follows.

Definition 3.1. The *O-strict fragment of RML* consists of terms of the form

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

where the type classes Θ_i are described below.

$$\begin{array}{lll} \Theta_0 & ::= & \mathsf{unit} \mid \mathsf{int} \\ \Theta_1 & ::= & \Theta_0 \mid \Theta_0 \to \Theta_1 \mid \mathsf{int\ ref} \end{array} \qquad \begin{array}{lll} \Theta_2 & ::= & \Theta_0 \mid \Theta_1 \to \Theta_0 \mid \mathsf{int\ ref} \\ \Theta_3 & ::= & \Theta_0 \mid \Theta_2 \to \Theta_3 \mid \mathsf{int\ ref} \end{array}$$

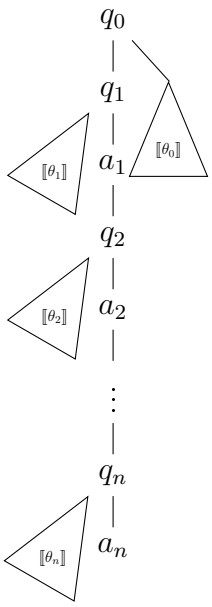


Figure 3.3: Prearena for $[\![\theta_1 \to \theta_2 \to \ldots \to \theta_n \to \beta \vdash \theta_0]\!]$

3.1.3 Examples of O-Strict Terms

While the definition of the O-Strict fragment may sound restrictive, it is actually a surprisingly expressive fragment and contains many difficult examples from the literature. For example, it includes call-by-value variants of the No Snapback and Scope Extrusion terms from Example 2.1 and Example 2.2.

Example 3.1 (No Snapback and Scope Extrusion)**.**

$$p : (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit} \vdash$$
$$\mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}\, p(\lambda y.x := 1); \mathbf{if}\, !x = 1\, \mathbf{then}\, \Omega\, \mathbf{else}\, () \cong p(\lambda y.\Omega)$$

$$M_1 \equiv \lambda F^{(\mathsf{int}\to\mathsf{int})\to\mathsf{int}}.\mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}\, F\, (\lambda y^{\mathsf{int}}.\mathbf{if}\, !x = 0\, \mathbf{then}\, x := y\, \mathbf{else}\, x := y - 1;\, !x)$$

$$M_2 \equiv \lambda F^{(\mathsf{int}\to\mathsf{int})\to\mathsf{int}}.F\, (\lambda y^{\mathsf{int}}.\mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}\, \mathbf{if}\, !x = 0\, \mathbf{then}\, x := y\, \mathbf{else}\, x := y - 1;\, !x)$$

$$M_3 \equiv \lambda F^{(\mathsf{int}\to\mathsf{int})\to\mathsf{int}}.F\, (\lambda y^{\mathsf{int}}.y)$$

As in the call-by-name case we have that $M_1 \ncong M_2 \cong M_3$.

The O-strict fragment also contains programs which contain complicated higher-order types and terms which differ only in their use of binders (this will show up in the game semantics as differing locations of P-pointers) such as the following terms.

- $\left\{ \begin{array}{l} f : ((\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \to \mathsf{unit} \vdash f(\lambda x^{\mathsf{unit}\to\mathsf{unit}}.f(\lambda y^{\mathsf{unit}\to\mathsf{unit}}.x())) : \mathsf{unit} \\ f : ((\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \to \mathsf{unit} \vdash f(\lambda x^{\mathsf{unit}\to\mathsf{unit}}.f(\lambda y^{\mathsf{unit}\to\mathsf{unit}}.y())) : \mathsf{unit} \end{array} \right.$
- $\left\{ \begin{array}{l} f : \mathsf{unit} \to \mathsf{unit} \to \mathsf{unit} \vdash \mathbf{let}\, g = f()\, \mathbf{in}\, (\mathbf{let}\, h = f()\, \mathbf{in}\, g()) : \mathsf{unit} \\ f : \mathsf{unit} \to \mathsf{unit} \to \mathsf{unit} \vdash \mathbf{let}\, g = f()\, \mathbf{in}\, (\mathbf{let}\, h = f()\, \mathbf{in}\, h()) : \mathsf{unit} \end{array} \right.$

The fragment includes several examples from the literature which are known to be challenging to verify.

Example 3.2. The following have been analysed respectively by Pitts and Stark [90], Ahmed et al. [9] and Stark [101].

(i) $\mathbf{let}\, c = \mathbf{ref}\, 0\, \mathbf{in}\, \lambda f^{\mathsf{unit}\to\mathsf{unit}}.(c := 1; f(); !c) \;\cong\; \lambda f^{\mathsf{unit}\to\mathsf{unit}}.(f(); 1)$

(ii) $\mathbf{let}\, c = \mathbf{ref}\, 0\, \mathbf{in}\, \lambda f^{\mathsf{unit}\to\mathsf{unit}}.(c := 0; f(); c := 1; f(); !c) \;\cong\; \lambda f^{\mathsf{unit}\to\mathsf{unit}}.(f(); f(); 1)$

(iii) $\mathbf{let}\, a = \mathbf{ref}\, 0\, \mathbf{in}\, \mathbf{let}\, r = \mathbf{ref}\, 0\, \mathbf{in}\, \lambda f.(r := !r+1; a := f(!r); r := !r-1; !a) \ncong \lambda f.f(1)$

The two equivalences, plus the No Snapback equivalence which appears in [17], are known to be extremely tricky to prove using methods based on logical relations. All three occurred in the literature as examples which could not be proved using particular techniques (although new methods which can handle them have since been developed [36]). The inequivalence is a somewhat surprising example which requires a rather delicate context to separate the terms.

Additionally, all of the imperative features of RML are included, allowing complex ground-type programs to be considered.

3.2 Decidability of the O-Strict Fragment

Having determined the largest fragment of RML for which O-pointers can be uniquely reconstructed, we can now turn to showing that observational equivalence is decidable for this fragment. It is already shown in [73] that finite automata are not powerful enough to represent the strategies we need. For example, the complete plays of the strategy $[\![\vdash \lambda f : (\mathsf{unit} \to \mathsf{unit}).f()]\!]$ are described by $q_0 \cdot a_0 \cdot X$ where $X = (q_1 \cdot q_2 \cdot X \cdot a_2 \cdot a_1 \cdot X) + \epsilon$. This cannot be recognised by a finite automaton. However, it is exactly the sort language VPA can capture and so we intend to encode O-strict terms as VPA.

3.2.1 P-Pointers

We aim to show decidability by representing the strategy denotations of O-strict terms as VPA. One hurdle we need to deal with is that while O-pointers can be safely omitted, pointers from P-moves can still be ambiguous. We need to encode their location if we wish to use VPA to recognise strategies.

In call-by-name algorithmic game semantics, one method of modelling pointers is to tag the appropriate moves with an index which identifies which of the potential justifiers is to be used. For example, the play $q_0\ q_1\ q_2\ q_1\ q_2\ q_3$, which arises from the term $f \vdash f(\lambda x.f(\lambda y.y))$, can be represented as $q_0\, q_1\, q_2\, q_1\, q_2\, q_3^2$ to indicate that q_3 is justified by the occurrence of q_2 which is itself justified by the second open occurrence of a q_1 which could potentially hereditarily justify third-order questions. Since each q_1 can justify at most one q_2 in the P-view this is sufficient to determine q_3's justifier. The only way to have multiple open q_1's is to have multiple open calls to second-order identifiers. In this example the first call to f causes a further, nested call of f to be made. That is, the maximum index we can require is the maximum level of nested calls of second-order identifiers present in our term. If we were looking at the term $f \vdash f(\lambda x.x)$ (which has the same type), there is only a single call and so the maximum index we would need is 1. However, the term $f \vdash f(\lambda x.f(\lambda y.f(\lambda x.x)))$ contains three nested calls, allowing three occurrences of q_1 to be open and requiring a maximum index of three. Despite the fact that at this type arbitrarily high indices can be needed, any given term contains only a finite number of nested calls (since we consider recursion free terms). This means that for a particular term we require only a finite number of indices. Unfortunately, this is no longer true in the call-by-value model. Consider the program below (which is O-strict).

$$
\begin{array}{l}
f : \mathsf{unit} \to \mathsf{int}, h : \mathsf{unit} \to \mathsf{unit} \to \mathsf{unit} \vdash \\
\textbf{while}\, f()\, \textbf{do} \\
\quad \textbf{let}\, g = h()\, \textbf{in}\, (); \\
\textbf{let}\, \hat{g} = h()\, \textbf{in} \\
\textbf{while}\, f()\, \textbf{do} \\
\quad \textbf{let}\, g = h()\, \textbf{in}\, (); \\
\hat{g}()
\end{array}
$$

This gives rise to plays of the form

$$q\ (q_f\ 1_f\ q_{1h}\ a_{1h})^*\ q_f\ 0_f\ q_{1h}\ a_{1h}\ (q_f\ 1_f\ q_{1h}\ a_{1h})^*\ q_f\ 0_f\ q_{2h}.$$

The two $(\overbrace{q_f\ 1_f\ q_{1h}\ a_{1h}})^*$ sections correspond to the two loops and can be unboundedly long. The final move, q_{2h}, is justified by the a_{1h} corresponding to $\hat{g}$. However, there can be an unbounded number of different occurrences of a_{1h} both between the two moves and between this a_{1h} and the start of the play. All of these moves are visible. Hence, using indices to indicate which justifier to use would seem to require an unbounded number of indices, which seems incompatible with recognition by VPA with finite alphabets.

An alternative method used in [76, 81] is to insert additional moves into the play to count back to the correct justifier. As we have seen there can be an unbounded number of potential justifiers between a move and its actual justifier, so counting back in this way using a VPA would seem to require using the stack. Unfortunately, we will need to use the stack to enforce the bracketing condition, as a play can contain an unbounded number of open questions and we need to ensure each one is answered exactly once before a play is considered complete. Hence, the size of the stack will need to be (roughly) proportional to the number of open questions. In [76, 81] this is not a problem as the number of open questions between a move and its justifier is sufficient to uniquely determine the location of the pointer. By contrast, in RML this is no longer the case. In the example above, the only open question is the initial move, so this approach does not seem to work either.

As existing approaches do not seem to suffice, we use an alternate method. Instead of trying to represent all the pointers present in a play, we concentrate only on representing the position of a single pointer. However, our automata will accept all strings representing the position of a pointer in an accepted play. So even though each individual word the automaton accepts may not have enough information to fully reconstruct the play, when we consider the full language we will be able to uniquely place all the justification pointers.

If $s\,m\,s'\,n\,s''$ is a sequence of moves, we will use $s\ \overset{\bullet}{m}\ s'\ \overset{\circ}{n}\ s''$ to represent that there is a pointer from n to m. We refer to moves tagged with $\bullet$ as target-moves and those tagged with $\circ$ as source-moves. When representing a strategy σ we will construct an automaton $\mathcal{A}_\sigma$ which accept all strings that are either the underlying move sequence of a complete play in σ, or the underlying move sequence plus the encoding of a single justification pointer from a P-question (in fact we will not encode the location of P-pointers which point at the initial move as the initial move is unique). Since all justification pointers from P-questions must have a representation in the automaton's language, this is sufficient to ensure that $\mathcal{L}(\mathcal{A}_\sigma) = \mathcal{L}(\mathcal{A}_\tau)$ if and only if $\mathbf{comp}(\sigma) = \mathbf{comp}(\tau)$.

Example 3.3. Consider the term below.

$$f : \mathsf{unit} \to \mathsf{unit} \to \mathsf{unit} \vdash \mathbf{let}\, g = f()\, \mathbf{in}\, \mathbf{let}\, h = f()\, \mathbf{in}\, g(); h()$$

The only complete play in the strategy denoting this term is (showing only ambiguous pointers) $q_0\ q_1\ a_1\ q_1\ a_1\ q_2\ a_2\ q_2\ a_2\ a_0$. Under our single-pointer representation, this strategy is represented by the language containing the three strings

1. $q_0 \cdot q_1 \cdot a_1 \cdot q_1 \cdot a_1 \cdot q_2 \cdot a_2 \cdot q_2 \cdot a_2 \cdot a_0$ (the underlying move sequence),
2. $q_0 \cdot q_1 \cdot \overset{\bullet}{a_1} \cdot q_1 \cdot a_1 \cdot \overset{\circ}{q_2} \cdot a_2 \cdot q_2 \cdot a_2 \cdot a_0$ (encoding the first pointer),
3. and $q_0 \cdot q_1 \cdot a_1 \cdot q_1 \cdot \overset{\bullet}{a_1} \cdot q_2 \cdot a_2 \cdot \overset{\circ}{q_2} \cdot a_2 \cdot a_0$ (encoding the second pointer).

Considering any one of these strings on their own is not sufficient to uniquely determine the play. For example, the first and second strings would also be included in our representation of

$$f : \mathsf{unit} \to \mathsf{unit} \to \mathsf{unit} \vdash \mathbf{let}\, g = f()\, \mathbf{in}\, \mathbf{let}\, h = f()\, \mathbf{in}\, g(); g()$$

whereas the first and third would be in the language for

$$f : \mathsf{unit} \to \mathsf{unit} \to \mathsf{unit} \vdash \mathbf{let}\, g = f()\, \mathbf{in}\, \mathbf{let}\, h = f()\, \mathbf{in}\, h(); h().$$

However, considering all three strings together allows us to tell which strategy we are representing and differentiate between these three terms.

Remark. Our single-pointer representation is not the only possible way of encoding P-pointers using a finite alphabet. For example, we could equally have chosen to insist that all P-moves whose justification pointers have the same target are encoded in the same word. We also choose to encode the location of pointers from all P-questions (except those pointing at the initial move since they are always uniquely determined and it will prove convenient to omit the initial move from plays when we construct our VPA) regardless of whether their location is actually ambiguous. We hope our choice gives an encoding which is both straightforward to understand and simple to implement.

3.2.2 Canonical Forms

In the decidability results for IA, the automata were constructed inductively over the β-normal forms of the language. Only having to consider β-normal forms made the constructions simpler. Unfortunately, in a call-by-value setting β-reduction no longer preserves observational equivalence. For example, $(\lambda x.())\,\Omega \not\cong ()$. However, there is a different notion of canonical form which is appropriate.

Definition 3.2. The canonical forms of RML are defined by the grammar below.

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

Proposition 3.3. *For every RML-term $\Gamma \vdash M : \theta$ there exists an RML-term $\Gamma \vdash N : \theta$ in canonical form, effectively constructible from M, such that $[\![\Gamma \vdash M]\!] = [\![\Gamma \vdash N]\!]$.*

We prove two auxiliary results first, which are special cases of the Proposition.

Lemma 3.1. *Any identifier x^{θ} satisfies Proposition 3.3. Moreover, the canonical form is of the form $\lambda y_1^{\theta}.\mathbb{C}$ when $\theta \equiv \theta_1 \to \theta_2$ and of the shape $\mathbf{mkvar}(\lambda y^{\mathsf{unit}}.\mathbb{C}, \lambda z^{\mathsf{int}}.\mathbb{C})$ if $\theta \equiv \mathsf{int\ ref}$.*

Proof. Induction with respect to the structure of θ.

- If θ is a base type, x^{θ} is already in canonical form. $x^{\mathsf{int\ ref}}$ can be converted to one using the rule

$$x^{\mathsf{int\ ref}} \longrightarrow \mathbf{mkvar}(\lambda u^{\mathsf{unit}}.!x, \lambda v^{\mathsf{int}}.x := v).$$

- For $\theta \equiv \theta_1 \to \theta_2$ we use the rule

$$x^{\theta_1 \to \theta_2} \longrightarrow \lambda z^{\theta_1}.\mathbf{let}\, v^{\theta_2} = xz\, \mathbf{in}\, v$$

  and appeal to the inductive hypothesis for z^{θ_1} and v^{θ_2}.

□

Lemma 3.2. *Suppose C_1, C_2 are canonical forms. Then $\mathbf{let}\, y^{\theta} = C_1\, \mathbf{in}\, C_2$, if typable, satisfies Proposition 3.3.*

Proof. Induction with respect to the structure of θ. If θ is a base type, the term is already in canonical form. If θ is not a base type, C_1 can take one of the following four shapes: $\mathbf{mkvar}(\lambda x^{\mathsf{unit}}.\mathbb{C}, \lambda y^{\mathsf{int}}.\mathbb{C})$, $\lambda x_1^{\theta_1}.\mathbb{C}$, $\mathbf{if}\ x^{\beta}\ \mathbf{then}\ \mathbb{C}\ \mathbf{else}\ \mathbb{C}$ or $\mathbf{let}\ \cdots\ \mathbf{in}\ \mathbb{C}$. We focus on the first two, to which the remaining cases will be reduced later.

- Suppose $C_1 \equiv \mathbf{mkvar}(\lambda x_1^{\mathsf{unit}}.C_{11}, \lambda x_2^{\mathsf{int}}.C_{12})$. Then $\theta \equiv \mathsf{int\ ref}$. Since C_2 is in canonical form, y can only occur in it as part of a canonical subterm of the form $y^{\mathsf{int\ ref}} := z^{\mathsf{int}}$ or $!y$. Hence, substitution for y creates non-canonical subterms of the shape $\mathbf{mkvar}(\lambda x_1^{\mathsf{unit}}.C_{11}, \lambda x_2^{\mathsf{int}}.C_{12}) := z$ and $!(\mathbf{mkvar}(\lambda x_1^{\mathsf{unit}}.C_{11}, \lambda x_2^{\mathsf{int}}.C_{12}))$. Using the rules

$$\begin{array}{rcl} !\mathbf{mkvar}(\lambda u^{\mathsf{unit}}.D_1, \lambda v^{\mathsf{int}}.D_2) & \longrightarrow & \mathbf{let}\, u = ()\, \mathbf{in}\, D_1 \\ \mathbf{mkvar}(\lambda u^{\mathsf{unit}}.D_1, \lambda v^{\mathsf{int}}.D_2) := z & \longrightarrow & D_2[z/v] \end{array}$$

  we can easily convert them (and thus the whole term) to canonical form.

- Suppose $C_1 \equiv \lambda x_1^{\theta_1}.C_3$ and $\theta \equiv \theta_1 \to \theta_2$. Let us substitute C_1 for the rightmost occurrence of y in C_2. This will create a non-canonical subterm in C_2 of the form $\mathbf{let}\, x^{\theta_2} = (\lambda x_1^{\theta_1}.C_3)C_4\, \mathbf{in}\, C_5 \equiv \mathbf{let}\, x^{\theta_2} = (\mathbf{let}\, x_1^{\theta_1} = C_4\, \mathbf{in}\, C_3)\, \mathbf{in}\, C_5$. By inductive hypothesis for θ_1, $\mathbf{let}\, x_1^{\theta_1} = C_4\, \mathbf{in}\, C_3$ can be converted to canonical form, say, C_6. Consequently, the non-canonical subterm $\mathbf{let}\, x^{\theta_2} = (\lambda x_1^{\theta_1}.C_3)C_4\, \mathbf{in}\, C_5$ can be transformed into the form $\mathbf{let}\, x^{\theta_2} = C_6\, \mathbf{in}\, C_5$, which — by inductive hypothesis for θ_2 — can also be converted to canonical form. Thus, we have shown how to recover canonical forms after substitution for the rightmost occurrence of y. Because of the choice of the rightmost occurrence, the transformation does not involve terms containing other occurrences of y, so it will also decrease their overall number in C_2 by one. Consequently, by repeated substitution for rightmost occurrences one can eventually arrive at a canonical form for $\mathbf{let}\, y^{\theta} = (\lambda x_1^{\theta_1}.C_3)\, \mathbf{in}\, C_2$.

For the remaining two cases it suffices to take advantage of the following conversions before referring to the two cases above.

$$\begin{array}{l} \mathbf{let}\, y = (\mathbf{if}\, x\, \mathbf{then}\, D_1\, \mathbf{else}\, D_0)\, \mathbf{in}\, E \longrightarrow \\ \qquad \mathbf{if}\, x\, \mathbf{then}\, (\mathbf{let}\, y = D_1\, \mathbf{in}\, E)\, \mathbf{else}\, (\mathbf{let}\, y = D_0\, \mathbf{in}\, E) \end{array}$$

$$\mathbf{let}\, y = (\mathbf{let}\, x = D\, \mathbf{in}\, E)\, \mathbf{in}\, F \longrightarrow \mathbf{let}\, x = D\, \mathbf{in}\, (\mathbf{let}\, y = E\, \mathbf{in}\, F)$$

□

Now we are ready to prove Proposition 3.3 by induction on term structure. The base cases of $()$ and i are trivial. That of x^θ follows from Lemma 3.1. The cases of $\lambda x^\theta.M$ and $\mathbf{while}\, M\, \mathbf{do}\, N$ follow directly from the inductive hypothesis. The cases of $\mathbf{succ}(M)$, $\mathbf{pred}(M)$, $\mathbf{if}\, M\, \mathbf{then}\, N_1\, \mathbf{else}\, N_0$ and $\mathbf{ref}\, M$ are only slightly more difficult. After invoking the inductive hypothesis one needs to apply the rules given below (and in the last case convert the occurrence of the $\mathsf{int\ ref}$-typed variable to canonical form using Lemma 3.1). Note that the let-bindings are all either of base type or of the form $\mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}\, M$.

$$\begin{array}{rcl}
\mathbf{succ}(D) & \longrightarrow & \mathbf{let}\, x = D\, \mathbf{in}\, \mathbf{succ}(x) \\
\mathbf{pred}(D) & \longrightarrow & \mathbf{let}\, x = D\, \mathbf{in}\, \mathbf{pred}(x) \\
\mathbf{if}\, D\, \mathbf{then}\, D_1\, \mathbf{else}\, D_0 & \longrightarrow & \mathbf{let}\, x = D\, \mathbf{in}\, (\mathbf{if}\, x\, \mathbf{then}\, D_1\, \mathbf{else}\, D_0) \\
\mathbf{ref}\, D & \longrightarrow & \mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}\, (\mathbf{let}\, y = D\, \mathbf{in}\, (\mathbf{let}\, z = x := y\, \mathbf{in}\, x))
\end{array}$$

For $!M$ and $M := N$ we take advantage of the fact that a canonical form of type $\mathsf{int\ ref}$ can only take three shapes: $\mathbf{mkvar}(\lambda x^{\mathsf{unit}}.\mathbb{C}, \lambda y^{\mathsf{int}}.\mathbb{C})$, $\mathbf{if}\, x^\beta\, \mathbf{then}\, \mathbb{C}\, \mathbf{else}\, \mathbb{C}$ or $\mathbf{let}\, \cdots\, \mathbf{in}\, \mathbb{C}$. An appeal to the inductive hypothesis for M and N and the conversions given below will then yield the canonical forms for $!M$ and $M := N$.

$$\begin{array}{rcl}
!\mathbf{mkvar}(\lambda u^{\mathsf{unit}}.D_1, \lambda v^{\mathsf{int}}.D_2) & \longrightarrow & \mathbf{let}\, u = ()\, \mathbf{in}\, D_1 \\
\mathbf{mkvar}(\lambda u^{\mathsf{unit}}.D_1, \lambda v^{\mathsf{int}}.D_2) := E & \longrightarrow & \mathbf{let}\, v = E\, \mathbf{in}\, D_2 \\
!(\mathbf{if}\, x\, \mathbf{then}\, D_1\, \mathbf{else}\, D_0) & \longrightarrow & \mathbf{if}\, x\, \mathbf{then}\, !D_1\, \mathbf{else}\, !D_0 \\
(\mathbf{if}\, x\, \mathbf{then}\, D_1\, \mathbf{else}\, D_0) := D & \longrightarrow & \mathbf{if}\, x\, \mathbf{then}\, (D_1 := D)\, \mathbf{else}\, (D_0 := D) \\
!(\mathbf{let}\, x = D\, \mathbf{in}\, E) & \longrightarrow & \mathbf{let}\, x = D\, \mathbf{in}\, !E \\
(\mathbf{let}\, x = D\, \mathbf{in}\, E) := F & \longrightarrow & \mathbf{let}\, x = D\, \mathbf{in}\, (E := F)
\end{array}$$

To convert $\mathbf{mkvar}(M, N)$ to canonical form we observe that a canonical form of function type can only take three shapes: $\lambda x^\theta.\mathbb{C}$, $\mathbf{if}\, x^\beta\, \mathbf{then}\, \mathbb{C}\, \mathbf{else}\, \mathbb{C}$ or $\mathbf{let}\, \cdots\, \mathbf{in}\, \mathbb{C}$. Hence, by appealing to the inductive hypothesis and then repeatedly applying the rules below we will arrive at a canonical form.

$$\begin{array}{rcl}
\mathbf{mkvar}(\mathbf{let}\, x = M\, \mathbf{in}\, D, E) & \longrightarrow & \mathbf{let}\, x = M\, \mathbf{in}\, \mathbf{mkvar}(D, E) \\
\mathbf{mkvar}(\mathbf{if}\, x\, \mathbf{then}\, D_1\, \mathbf{else}\, D_0, E) & \longrightarrow & \mathbf{if}\, x\, \mathbf{then}\, \mathbf{mkvar}(D_1, E)\, \mathbf{else}\, \mathbf{mkvar}(D_0, E) \\
\mathbf{mkvar}(\lambda u^{\mathsf{unit}}.D, \mathbf{let}\, x = M\, \mathbf{in}\, E) & \longrightarrow & \mathbf{let}\, x = M\, \mathbf{in}\, \mathbf{mkvar}(\lambda u^{\mathsf{unit}}.D, E) \\
\mathbf{mkvar}(\lambda u^{\mathsf{unit}}.D, \mathbf{if}\, x\, \mathbf{then}\, E_1\, \mathbf{else}\, E_0) & \longrightarrow & \\
\multicolumn{3}{r}{\mathbf{if}\, x\, \mathbf{then}\, \mathbf{mkvar}(\lambda u^{\mathsf{unit}}.D, E_1)\, \mathbf{else}\, \mathbf{mkvar}(\lambda u^{\mathsf{unit}}.D, E_0)}
\end{array}$$

Finally, we handle application MN. First we apply the inductive hypothesis to both terms. Then we use the rules below to reveal the λ-abstraction inside the

canonical form of M.

$$\begin{array}{rcl}(\mathbf{if}\, x\, \mathbf{then}\, D_1\, \mathbf{else}\, D_0)E & \longrightarrow & \mathbf{if}\, x\, \mathbf{then}\, (D_1E)\, \mathbf{else}\, (D_0E) \\ (\mathbf{let}\, x = D\, \mathbf{in}\, E)F & \longrightarrow & \mathbf{let}\, x = D\, \mathbf{in}\, (EF)\end{array}$$

Now it suffices to be able to deal with terms of the form $(\lambda x^{\theta}.C_1)C_2 \equiv \mathbf{let}\, x = C_2\, \mathbf{in}\, C_1$ and this is exactly what Lemma 3.2 does.

All our transformations preserve denotations. This is straightforward to check using the axioms resulting from Moggi's monadic approach to modelling call-by-value languages [71]. □

3.2.3 Construction

We can now construct VPA which represent $\mathsf{RML}_{\text{O-Str}}$-terms. To simplify our constructions we will omit the initial move from our plays and represent a strategy by a family of VPA indexed by the initial move. That is, if we define $[\![\ldots]\!]_i$ by $[\![\Gamma \vdash M : \theta]\!] = \sum_{i \in I_{[\![\Gamma]\!]}} i\, [\![\Gamma \vdash M : \theta]\!]_i$, then we will define automata $\mathcal{A}^i_{\Gamma \vdash M}$ which accept representations of all complete plays in $[\![M]\!]_i$. Throughout our constructions we will not enforce the constraint that at most one pointer is encoded in each run. This is simple to ensure at the end of the construction. However, we will make sure that a move cannot be tagged as a source-move unless the target of its justification pointer was previously tagged as a target-move. The alphabet of $\mathcal{A}^i_{\Gamma \vdash M:\theta}$ will contain all moves from the prearena $[\![\Gamma \vdash \theta]\!]$ except the initial moves, plus additional copies of P-questions and their justifiers tagged as source- and target-moves respectively. This gives alphabets which are finite and of size linear in the size of $\Gamma \vdash \theta$. We will partition the alphabets such that all P-questions are pushes, all O-answers pops and everything else noops. When describing our constructions we may omit the stack information from the transitions if it aids clarity. We will further insist that if we include a transition $s_1 \overset{\mathring{m}/\gamma}{\rightarrow} s_2$, then we must also have a transition $s_1 \overset{m/\gamma}{\rightarrow} s_2$ (i.e. the tagged and untagged transitions both push the same symbol and have the same destination). Due to determinacy of the strategies we are representing, such states with outgoing source-transitions will be the only P-states in which more than one transition can be taken.

Our construction proceeds inductively over the canonical forms of $\mathsf{RML}_{\text{O-Str}}$, so we consider each case in turn.

3.2.3.1 Simple Cases

The denotations of several of the canonical forms can be described using regular expressions.

- $\mathcal{L}(\mathcal{A}^i_{()}) = \bullet$.
- $\mathcal{L}(\mathcal{A}^i_j) = j$.
- $\mathcal{L}(\mathcal{A}^i_{x^\beta}) = \pi_x(i)$ where π_x is a suitable projection function which picks out the component of the initial move associated with x.
- $\mathcal{L}(\mathcal{A}^i_{\mathbf{succ}(x^\beta)}) = \pi_x(i) \oplus 1$ and $\mathcal{L}(\mathcal{A}^i_{\mathbf{pred}(x^\beta)}) = \pi_x(i) \ominus 1$ where π_x is a suitable projection function and $\oplus, \ominus$ are addition and subtraction modulo the size of int.
- $\mathcal{L}(\mathcal{A}^i_{x^{\mathsf{int\ ref}} := y^{\mathsf{int}}}) = \mathsf{write}(\pi_y(i))_x\, \mathsf{ok}_x\, \bullet$.
- $\mathcal{L}(\mathcal{A}^i_{!x^{\mathsf{int\ ref}}}) = \sum_{j \in \mathsf{int}} \mathsf{read}_x\, j_x\, j$.

Conditionals simply act as one of their branches.

- $\mathcal{A}^i_{\mathbf{if}\, x^\beta\, \mathbf{then}\, M\, \mathbf{else}\, N} = \begin{cases} \mathcal{A}^i_M & \text{if } \pi_x(i) > 0 \\ \mathcal{A}^i_N & \text{if } \pi_x(i) = 0 \end{cases}$.

3.2.3.2 Relatively Simple Cases

Several other cases can be handled by simple modifications of the automata for their subterms. In all of these cases we do not need to worry about justification pointers as long as we preserve those encoded by the sub-automata.

- $\mathbf{mkvar}(\lambda x^{\mathsf{unit}}.M, \lambda y^{\mathsf{int}}.N)$
  - The bad-variable constructor takes a read-function and a write-function and combines them together to form an $\mathsf{int\ ref}$. In the game-semantic model this will combine the two sets of plays and rename them into the $\mathsf{int\ ref}$-arena. Due to the switching condition we cannot start a new play from $[\![M]\!]$ or $[\![N]\!]$ until the previous one has finished.
  - Let $L_M = \mathcal{L}(\mathcal{A}^{(i,\bullet)}_{\Gamma, x:\mathsf{unit} \vdash M:\mathsf{int}})$ and $L^j_N = \mathcal{L}(\mathcal{A}^{(i,j)}_{\Gamma, y:\mathsf{int} \vdash N:\mathsf{unit}})[\bullet \backslash \mathsf{ok}]$ (that is the language accepted by the automaton for $[\![N]\!]_{(i,j)}$ but with the final move renamed to ok instead of the singleton move from $[\![\mathsf{unit}]\!]$, $\bullet$). Then

$$\mathcal{L}(\mathcal{A}^i_{\mathbf{mkvar}(\lambda x^{\mathsf{unit}}.M, \lambda y^{\mathsf{int}}.N)}) = \bullet \cdot \left(\mathsf{read} \cdot L_M + \sum_j \mathsf{write}(j) \cdot L^j_N \right)^* .$$

- $\mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}\, M$

  - Constructing $\mathcal{A}^i_{\mathbf{let}\, x=\mathbf{ref}\, 0\, \mathbf{in}\, M}$ requires restricting the behaviour of x to that of a "good variable". To do this we take a copy of $\mathcal{A}^{(i,\bullet)}_{\Gamma,x:\mathsf{int}\ \mathsf{ref}\vdash M}$ for each $j \in \mathsf{int}$, denoting state s in the jth copy as s^j. The initial state will be the initial state from the 0th copy, whereas the final states in every copy will still be final. As stack symbols we just use those of $\mathcal{A}^{(i,\bullet)}_{\Gamma,x:\mathsf{int}\ \mathsf{ref}\vdash M}$.
  - All non-x-transitions are preserved. That is, if $s_1 \xrightarrow{m} s_2$ in $\mathcal{A}^{(i,\bullet)}_{\Gamma,x\vdash M}$ where m is not an x-move, then $s_1^j \xrightarrow{m} s_2^j$ for all $j \in \mathsf{int}$ (if the original transition is a push- or pop-transition then the new transitions will be too).
  - If $\mathcal{A}^{(i,\bullet)}_{\Gamma,x\vdash M}$ contains transitions $s_1 \xrightarrow{\mathsf{write}(k)_x/\gamma} s_2 \xrightarrow{\mathsf{ok}_x,\gamma} s_3$ then instead we add $s_1^j \xrightarrow{\epsilon} s_3^k$ for each $j \in \mathsf{int}$. Note that after $\mathsf{write}(k)_x$, due to visibility O's only legal response is to respond with ok_x.
  - If $\mathcal{A}^{(i,\bullet)}_{\Gamma,x\vdash M}$ contains transitions $s_1 \xrightarrow{\mathsf{read}_x/\gamma} s_2 \xrightarrow{j_x,\gamma} s_3$ then instead we add $s_1^j \xrightarrow{\epsilon} s_3^j$.
  - Once the construction is complete, if a state has an ϵ-transition then this will be the only transition out of that state (since read_x and $\mathsf{write}(k)_x$ were P-moves). This means we can easily compress the ϵ-transitions and maintain determinacy. That is, if $s_1 \xrightarrow{\epsilon}{}^* s_2 \xrightarrow{m} s_3$ then we can safely remove the ϵ-transitions and instead add $s_1 \xrightarrow{m} s_3$ without losing determinacy. Note also that if there is a cycle of ϵ-transitions then this represents divergent behaviour and we can just remove these transitions.

- $\mathbf{while}\, M\, \mathbf{do}\, N$

  Constructing $\mathcal{A}^i_{\mathbf{while}\, M\, \mathbf{do}\, N}$ requires a copy of both $\mathcal{A}^i_M$ and $\mathcal{A}^i_N$ (we take the disjoint union of both their states and stack symbols). The initial and final states will be those of $\mathcal{A}^i_M$. We denote the initial states of $\mathcal{A}^i_M$ and $\mathcal{A}^i_N$ by i_M and i_N respectively. All $[\![\Gamma]\!]$-transitions will be kept unchanged. All final transitions $s_1 \xrightarrow{j} s_2$ for $j > 0$ in $\mathcal{A}^i_M$ will be replaced by $s_1 \xrightarrow{\epsilon} i_N$, whereas $s_1 \xrightarrow{0} s_2$ will be changed to $s_1 \xrightarrow{\bullet} s_2$. Finally, if $\mathcal{A}^i_N$ contains a transition $s_1 \xrightarrow{\bullet} s_2$ then this is replaced by $s_1 \xrightarrow{\epsilon} i_M$. As in the previous case, if a state has an outgoing ϵ-transition then this will be the only transition out of this state so we can compress them out.

- $\mathbf{let}\, x^\beta = M\, \mathbf{in}\, N$

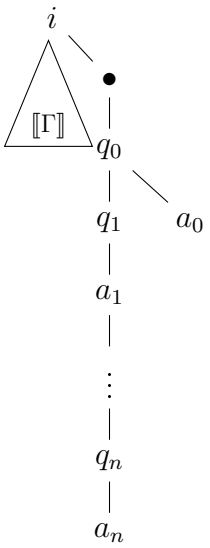


Figure 3.4: Prearena for $[\![\Gamma \vdash (\theta_0 \to \ldots \to \theta_0) \to \theta_0]\!]$

The game semantics of $\mathbf{let}\, x^\beta = M\, \mathbf{in}\, N$ is essentially a concatenation of a play from $[\![M]\!]$ and a play from $[\![N]\!]$.

To construct $\mathcal{A}^i_{\mathbf{let}\, x^\beta = M\, \mathbf{in}\, N}$, we take a copy of $\mathcal{A}^i_{\Gamma \vdash M}$ and a copy of $\mathcal{A}^{(i,j)}_{\Gamma, x \vdash N}$ for each $j \in M_{[\![\beta]\!]}$ (again taking the disjoint union of both their states and stack symbols). The initial state is the initial state from $\mathcal{A}^i_{\Gamma \vdash M}$, the final states are those of $\mathcal{A}^{(i,j)}_{\Gamma, x \vdash N}$ and all transitions are maintained unchanged, except that if $s_1 \xrightarrow{j} s_2$ in $\mathcal{A}^i_{\Gamma \vdash M}$ where j is a $[\![\beta]\!]$-answer, then instead $s_1 \xrightarrow{\epsilon} i^j_N$ where i^j_N is the initial state of $\mathcal{A}^{(i,j)}_{\Gamma, x \vdash N}$.

3.2.3.3 λ-abstraction

We now consider the trickier cases, starting with λ-abstraction. Since the conversion to canonical form preserves types, we will only have to deal with λ-abstractions of types of the form $\Theta_1 \to \Theta_0$. The relevant prearena is shown in Figure 3.4, where $n \geq 0$. The definition of the semantics of $[\![\Gamma \vdash \lambda x.M]\!]$ describes it as an interleaving of plays from $[\![\Gamma, x \vdash M]\!]$, each of which is started by O playing q_0. However, only O will switch between threads and visibility and the switching condition restrict this to occur only after P plays on the right-hand side (i.e. after $\bullet$, a_0 or some q_j). Further, since all of O's moves on the right-hand side except q_0 are answers, bracketing prevents O from returning to a previously opened $[\![M]\!]$-thread until all subsequently opened threads have been closed. That is, the semantics will actually consist of nested copies of plays from $[\![\Gamma, x \vdash M]\!]$. We will use the stack to keep track of this nesting and

ensure we only accept when the final thread is closed and all questions have been answered.

This is the first construction for which we will need to add pointer information[2]. We can rely on the automaton for $[\![\Gamma, x \vdash M]\!]$ to keep track of the pointers from $[\![\Gamma]\!]$-moves and also those from q_i moves for $i > 1$. However, in $[\![\Gamma, x \vdash M]\!]$ q_1 is justified by the initial move and so no pointer information will be recorded. In the prearena used for $[\![\Gamma \vdash \lambda x.M]\!]$ this is no longer the case, so we must include plays where q_1 and q_0 are tagged as source- and target-moves. To deal with this, as our set of states we will take:

- Four copies of the states of each $\mathcal{A}^{(i,q_0)}_{\Gamma,x:\theta_1 \vdash M:\theta_0}$ for each $q_0 \in I_{[\![\theta_1]\!]}$.
  - We write (s, q_0) for the first copy of state s from $\mathcal{A}^{(i,q_0)}_{\Gamma,x \vdash M}$ to indicate that this state belongs to a thread opened by q_0 (if x is of type int then q_0 represents multiple moves).
  - The second copy is needed to keep track of whether a thread was the first opened thread. This allows us to know when we have reached a complete play. We write $(s, q_0)'$ for the copy of state s from $\mathcal{A}^{(i,q_0)}_{\Gamma,x \vdash M}$ in the first opened thread.
  - Finally, we need an additional copy of both marked and unmarked states to allow us to encode pointers. We write $(s, \overset{\bullet}{q_0})$ or $(s, \overset{\bullet}{q_0})'$ to indicate that the q_0 which opened this thread was tagged as a target move. Whenever this thread can perform a P-question justified by q_0 (i.e. a q_1) we will need to allow it to be tagged as a source-move.
- We will also need three fresh states (1), the initial state, (2), the only final state, and (3) which will be reached whenever O could open a new $[\![M]\!]$-thread but the play is not complete.

As stack symbols we will use the disjoint union of the stack symbols of each $\mathcal{A}^{(i,q_0)}_{\Gamma,x:\theta_1 \vdash M:\theta_0}$ plus the states of the new automaton. Our transitions will be as follows.

- (1) $\xrightarrow{\bullet}$ (2).

[2]The location of the justification pointer from q_1 is always uniquely determined. However, for consistency our representation requires us to encode the location of pointers from all P-questions (except those pointing at the initial move).

- If the initial state of $\mathcal{A}^{(i,q_0)}_{\Gamma,x\vdash M}$ is j, then (2) $\xrightarrow{q_0}$ $(j,q_0)'$,(2) $\xrightarrow{\overset{\bullet}{q_0}}$ $(j,\overset{\bullet}{q_0})'$, (3) $\xrightarrow{q_0}$ (j,q_0) and (3) $\xrightarrow{\overset{\bullet}{q_0}}$ $(j,\overset{\bullet}{q_0})$. New threads can be opened from state (2) (when the play is complete) or state (3) (when P has played on the right-hand side but the play is incomplete). Threads opened from (2) are marked as being the first opened thread, whereas those opened from (3) are not. In both cases we must allow the opening move to be a target-move.

- If $\mathcal{A}^{(i,q_0)}_{\Gamma,x\vdash M}$ contains a transition $s_1 \xrightarrow{a_0} s_2$ then we instead have transitions $(s_1,q_0) \xrightarrow{a_0}$ (3),$(s_1,\overset{\bullet}{q_0}) \xrightarrow{a_0}$ (3), $(s_1,q_0)' \xrightarrow{a_0}$ (2) and $(s_1,\overset{\bullet}{q_0})' \xrightarrow{a_0}$ (2). When we close a thread we return to either (2) or (3) depending on whether the play is complete.

- Suppose $\mathcal{A}^{(i,q_0)}_{\Gamma,x\vdash M}$ contains transitions of the form $s_1 \xrightarrow{q_j/\gamma} s_2 \xrightarrow{a_j,\gamma} s_3$. Note that as s_1 is a P-state this will be the only transition out of s_1[3]. Due to visibility and the switching condition, the only transitions which can be taken immediately after q_j are on a_j's. There could be many different possible a_j's (if the appropriate part of the type is int or $\mathsf{int\ ref}$) but s_3 is uniquely determined by s_1 and a_j.

  Our new automaton will have transitions $(s_1,q_0) \xrightarrow{q_j/(s_1,q_0)}$ (3), $(s_1,q_0)' \xrightarrow{q_j/(s_1,q_0)'}$ (3),$(s_1,\overset{\bullet}{q_0}) \xrightarrow{q_j/(s_1,\overset{\bullet}{q_0})}$ (3) and $(s_1,\overset{\bullet}{q_0})' \xrightarrow{q_j/(s_1,\overset{\bullet}{q_0})'}$ (3). After q_j O has the option of starting a new thread.

  We also have the transitions (3) $\xrightarrow{a_j,(s_1,q_0)}$ (s_3,q_0), (3) $\xrightarrow{a_j,(s_1,q_0)'}$ $(s_3,q_0)'$,(3) $\xrightarrow{a_j,(s_1,\overset{\bullet}{q_0})}$ $(s_3,\overset{\bullet}{q_0})$ and (3) $\xrightarrow{a_j,(s_1,\overset{\bullet}{q_0})'}$ $(s_3,\overset{\bullet}{q_0})'$. This allows O to resume the pending thread.

  Additionally, if $j=1$ then we also have the transitions $(s_1,\overset{\bullet}{q_0}) \xrightarrow{\overset{\circ}{q_1}/(s_1,\overset{\bullet}{q_0})}$ (3) and $(s_1,\overset{\bullet}{q_0})' \xrightarrow{\overset{\circ}{q_1}/(s_1,\overset{\bullet}{q_0})'}$ (3) encoding a pointer.

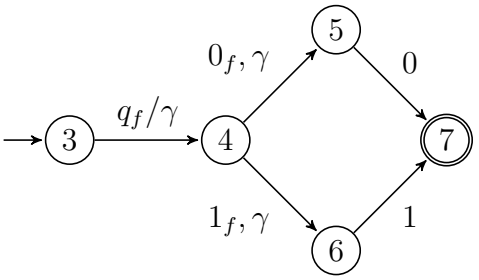
  Also, note that if the q_j or a_j in the original automaton were source- or target-moves then the transitions we create in the new automaton should be too.

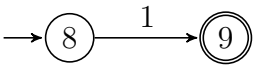
- Finally, all transitions on $[\![\Gamma]\!]$-moves are preserved unchanged. That is, if $\mathcal{A}^{(i,q_0)}_{\Gamma,x\vdash M}$ contains a transition $s_1 \xrightarrow{m} s_2$ where m is a $[\![\Gamma]\!]$-move, then we keep this transition in the forms $(s_1,q_0) \xrightarrow{m} (s_2,q_0)$, $(s_1,q_0)' \xrightarrow{m} (s_2,q_0)'$,$(s_1,\overset{\bullet}{q_0}) \xrightarrow{m}$ $(s_2,\overset{\bullet}{q_0})$ and $(s_1,\overset{\bullet}{q_0})' \xrightarrow{m} (s_2,\overset{\bullet}{q_0})'$. Here $s_1 \xrightarrow{m} s_2$ could be a push-, pop-, or a noop-transition and possibly a source- or target- move, in which case these

[3] Actually, due to our encoding of justification pointers, it is possible there will be two transitions out of s_1, one tagged as a source-transition and the other untagged. However, we insist that they will both push γ and move into s_2.

properties (and the pushed/popped symbol) should all be the same in the new automaton.

Example 3.4. Consider the term $f : \mathsf{unit} \to \mathsf{int} \vdash \lambda x : \mathsf{int}.\mathbf{if}\, x > 0\, \mathbf{then}\, 1\, \mathbf{else}\, f()$. For simplicity we will assume that int contains only 0 and 1. Our family of automata for $f, x \vdash \mathbf{if}\, x > 0\, \mathbf{then}\, 1\, \mathbf{else}\, f()$ consists of two members:

- $\mathcal{A}^{(\bullet_f, 0_x)}_{f,x \vdash \mathbf{if}\, x>0\, \mathbf{then}\, 1\, \mathbf{else}\, f()} =$ 

- $\mathcal{A}^{(\bullet_f, 1_x)}_{f,x \vdash \mathbf{if}\, x>0\, \mathbf{then}\, 1\, \mathbf{else}\, f()} =$ 

Applying our construction for λ-abstraction produces the VPA in Figure 3.5. Restricting this to allow only a single pointer to be represented (or in this case, as there will never be any pointers encoded, removing unmatched source-transitions) gives the VPA in Figure 3.6. In this case (and in any case where x has base type) there can be no nesting of threads or encoding of pointers. The abstraction simply iterates plays of the unabstracted strategies.

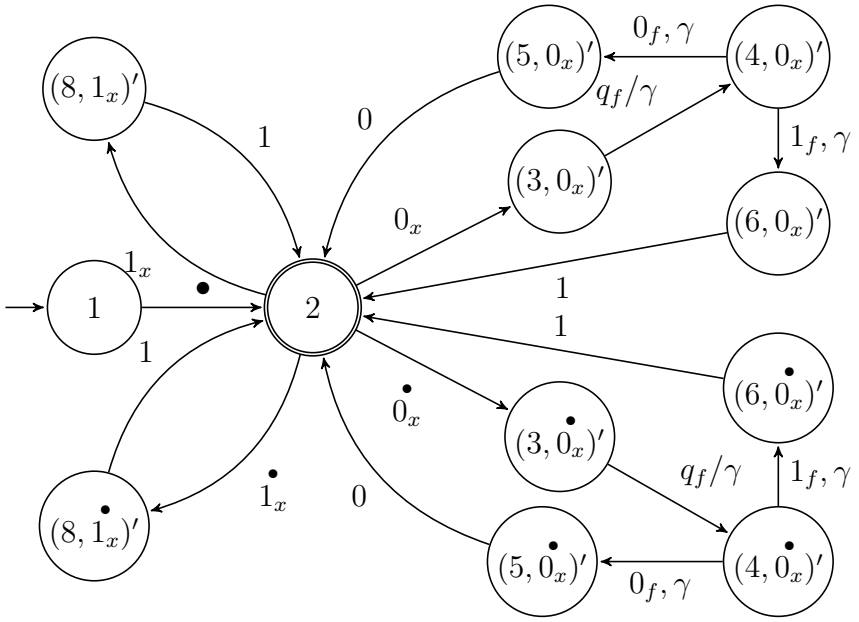


Figure 3.5: VPA for $f : \mathsf{unit} \to \mathsf{int} \vdash \lambda x : \mathsf{int}.\mathbf{if}\, x > 0\, \mathbf{then}\, 1\, \mathbf{else}\, f()$

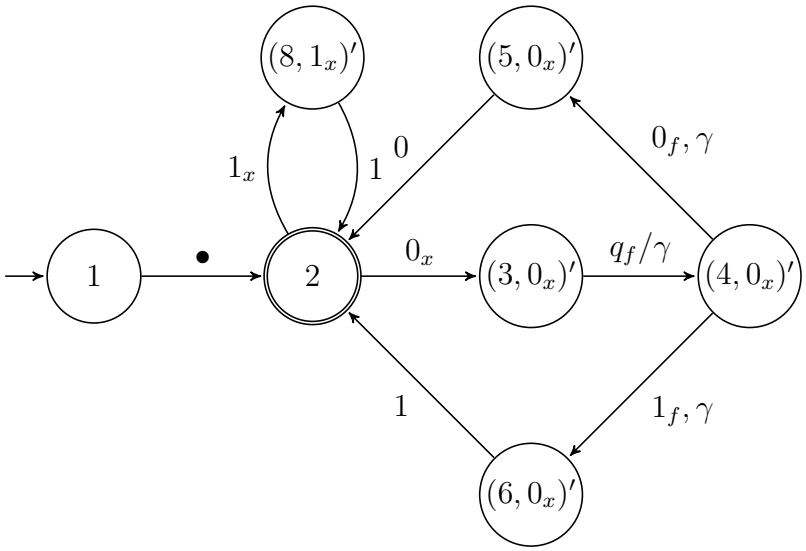


Figure 3.6: Simplified VPA for $f : \mathsf{unit} \to \mathsf{int} \vdash \lambda x : \mathsf{int}.\mathbf{if}\, x > 0\, \mathbf{then}\, 1\, \mathbf{else}\, f()$

Example 3.5. Now consider the term $\vdash \lambda f : \mathsf{unit} \to \mathsf{unit}.f()$. The family of automata for $f \vdash f()$ is a singleton family containing only the automaton below.

$$\mathcal{A}^q_{f \vdash f()} = \quad \to (4) \xrightarrow{q_f/\gamma} (5) \xrightarrow{a_f, \gamma} (6) \xrightarrow{a} ((7))$$

Applying our construction gives the automaton shown in Figure 3.7. The stack is used to keep track of open questions. Every time P plays q_f or a, we return to state 3, unless the play is complete in which case we enter the final state 2.

3.2.3.4 let x= z N in M

The remaining cases have the form $\Gamma, z : \theta_3 \vdash \mathbf{let}\, x = zN\, \mathbf{in}\, M : \theta_2$. We consider them in order of increasing complexity.

- $\mathbf{let}\, x = zy^\beta\, \mathbf{in}\, M$

  We must have $x : \theta_3$, $y : \theta_0$, $z : \theta_0 \to \theta_3$ and $M : \theta_2$, where each θ_i is a type from the corresponding Θ_i. Play will commence with P questioning z, using the value of y contained in the initial move. O must respond with an initial $[\![\theta_3]\!]$-move. Play then continues as it would in $\Gamma, x, y, z \vdash M$ on this initial move except with all x-moves renamed into z-moves.

  If there are any P-questions justified by the initial $[\![\theta_3]\!]$-move then we will have to encode their justification pointer location.

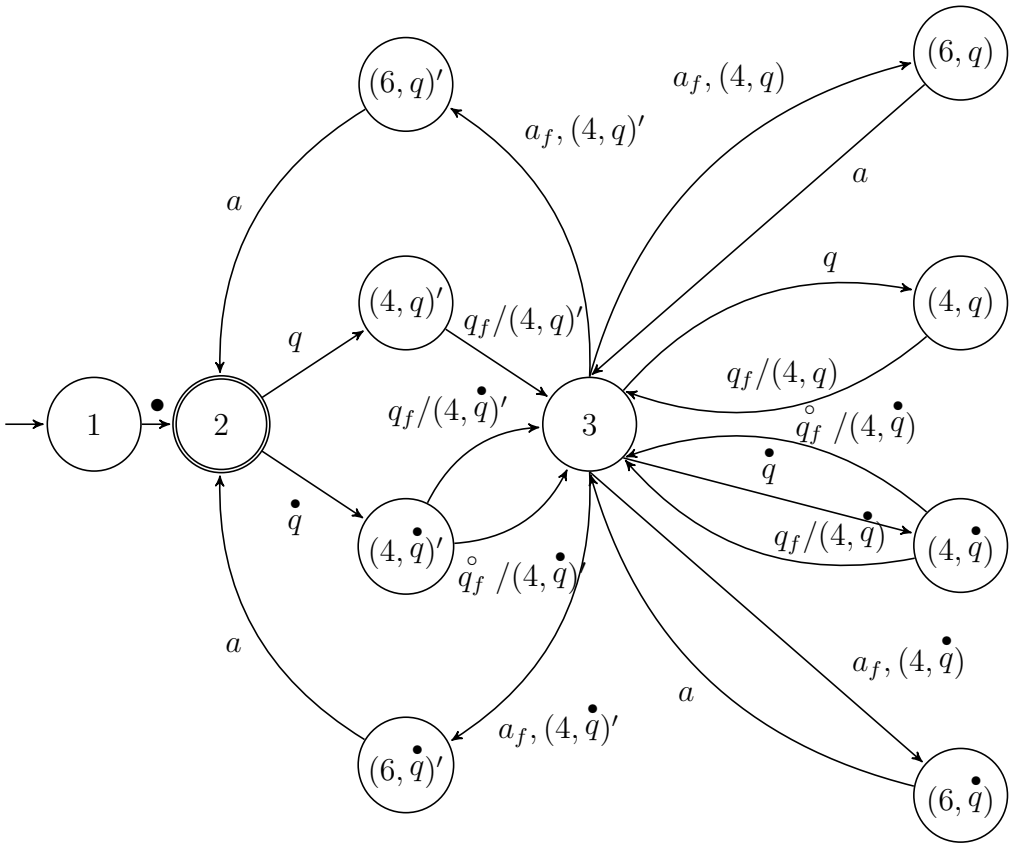


Figure 3.7: VPA for $\vdash \lambda f : \mathsf{unit} \to \mathsf{unit}.f()$

To construct $\mathcal{A}^{(i,\bullet,j)}_{\Gamma,z,y\vdash \mathbf{let}\, x=zy\,\mathbf{in}\, M}$, we will take two copies of $\mathcal{A}^{(i,\bullet,j,k)}_{\Gamma,z,y,x\vdash M}$ for each $k \in I_{[\![x]\!]}$. The second copy is required for encoding pointers as in the previous case. We will differentiate between the two copies by tagging state s in the second copy $\overset{\bullet}{s}$. The final states of these automata are preserved. We will also include two fresh states (1) and (2) with (1) being initial. We take as stack symbols the disjoint union of the stack symbols of each $\mathcal{A}^{(i,\bullet,j,k)}_{\Gamma,z,y,x\vdash M}$. Our transitions are then

- (1) $\xrightarrow{j_z}$ (2) where j_z is the copy of j (the initial y-move) in the z-component.
- For each $k \in I_{[\![x]\!]}$, (2) $\xrightarrow{k}$ s^k_M and (2) $\xrightarrow{\overset{\bullet}{k}}\overset{\bullet}{s^k_M}$ where s^k_M is the initial state of $\mathcal{A}^{(i,\bullet,j,k)}_{\Gamma,z,y,x\vdash M}$.
- All x-transitions are relabelled into z-transitions. That is, if $s \xrightarrow{m_x} t$ in $\mathcal{A}^{(i,\bullet,j,k)}_{\Gamma,z,y,x\vdash M}$ where m_x is a move from the x-component, then instead we have $s \xrightarrow{m_z} t$ and $\overset{\bullet}{s}\xrightarrow{m_z}\overset{\bullet}{t}$ where m_z is the appropriate relabelling. Additionally, if m_x was a P-move justified by the initial move, we include the transition $\overset{\bullet}{s}\xrightarrow{\overset{\circ}{m_z}}\overset{\bullet}{t}$.
- All other transitions are included unmodified. So if $s \xrightarrow{m} t$ in $\mathcal{A}^{(i,\bullet,j,k)}_{\Gamma,z,y,x\vdash M}$ where m is not from the x-component we have $s \xrightarrow{m} t$ and $\overset{\bullet}{s}\xrightarrow{m}\overset{\bullet}{t}$.

- $\mathbf{let}\, x = z\, \mathbf{mkvar}(\lambda u^{\mathsf{unit}}.M, \lambda v^{\mathsf{int}}.N)\, \mathbf{in}\, Q$

  This case is similar to the previous one. The complication is that after P plays $\bullet_z$ (the initial $\mathsf{int\ ref}$-move in the z-component) and whenever P plays in x, O can play read or $\mathsf{write}(n)$ which will start an $[\![M]\!]$- or $[\![N]\!]$-thread as appropriate. Control cannot leave the $[\![M]\!]$- or $[\![N]\!]$-thread until it is closed, whereupon play resumes in the same place it was before the thread was opened. Similarly to the previous case, to construct $\mathcal{A}^{(i,\bullet)}_{\Gamma,z\vdash \mathbf{let}\, x=z\, \mathbf{mkvar}(\lambda u^{\mathsf{unit}}.M,\lambda v^{\mathsf{int}}.N)\, \mathbf{in}\, Q}$, we will use two fresh states (1) and (2), with (1) being the initial state, and two copies of $\mathcal{A}^{(i,\bullet,j)}_{\Gamma,z,x\vdash Q}$ for each $j \in I_{[\![x]\!]}$ (as in previous cases the two copies are required for encoding pointers). The only final states will be the ones from these automata. Finally, we will also have distinct copies of $\mathcal{A}^{(i,\bullet,\bullet)}_{\Gamma,z,u\vdash M}$ and $\mathcal{A}^{(i,\bullet,k)}_{\Gamma,z,v\vdash N}$ for each $k \in \mathsf{int}$ and each state in

  $$\mathcal{S} = \{\,(2)\,\} \uplus \{\, r \mid (r = s \vee r = \overset{\bullet}{s}) \wedge t \xrightarrow{m_x} s \text{ in some } \mathcal{A}^{(i,\bullet,j)}_{\Gamma,z,x\vdash Q} \text{ with } m_x \text{ a P-}x\text{-move} \,\}.$$

  We will denote these states (s_1, s_2) where s_1 is either an $[\![M]\!]$- or an $[\![N]\!]$-state and $s_2 \in \mathcal{S}$. As stack symbols we take the disjoint union of the stack symbols from each $\mathcal{A}^{(i,\bullet,j)}_{\Gamma,z,x\vdash Q}$, $\mathcal{A}^{(i,\bullet,\bullet)}_{\Gamma,z,u\vdash M}$ and $\mathcal{A}^{(i,\bullet,k)}_{\Gamma,z,v\vdash N}$.

Transitions are as follows:

- (1) $\xrightarrow{\bullet_z}$ (2).
- (2) $\xrightarrow{j_z}$ s^j_Q and (2) $\xrightarrow{\overset{\bullet}{j_z}}\overset{\bullet}{s^j_Q}$ where $j \in I_{[\![x]\!]}$ and s^j_Q is the initial state of $\mathcal{A}^{(i,\bullet,j)}_{\Gamma,z,x\vdash Q}$.
- For each $s \in \mathcal{S}$ we have $s \xrightarrow{\mathsf{read}} (s_M, s)$ where s_M is the initial state of $\mathcal{A}^{(i,\bullet,\bullet)}_{\Gamma,z,u\vdash M}$. Similarly, for each $k \in \mathsf{int}$ and $s \in \mathcal{S}$ we have $s \xrightarrow{\mathsf{write}(k)} (s^k_N, s)$ where s^k_N is the initial state of $\mathcal{A}^{(i,\bullet,k)}_{\Gamma,z,v\vdash N}$.
- Transitions within $\mathcal{A}^{(i,\bullet,j)}_{\Gamma,z,x\vdash Q}$ are kept unchanged except that x-moves are relabelled as z-moves. If $s \xrightarrow{m_x} t$ in $\mathcal{A}^{(i,\bullet,j)}_{\Gamma,z,x\vdash Q}$ where m_x is a move from the x-component, then instead we have $s \xrightarrow{m_z} t$ and $\overset{\bullet}{s}\xrightarrow{m_z}\overset{\bullet}{t}$. Additionally, if m_x was a P-move justified by the initial move, we also include the transition $\overset{\bullet}{s}\xrightarrow{\overset{\circ}{m_z}}\overset{\bullet}{t}$. If $s \xrightarrow{m} t$ in $\mathcal{A}^{(i,\bullet,j)}_{\Gamma,z,x\vdash Q}$ where m is not from the x-component then we have $s \xrightarrow{m} t$ and $\overset{\bullet}{s}\xrightarrow{m}\overset{\bullet}{t}$.
- Transitions within any of the copies of $\mathcal{A}^{(i,\bullet,\bullet)}_{\Gamma,z,u\vdash M}$ or $\mathcal{A}^{(i,\bullet,k)}_{\Gamma,z,v\vdash N}$ are also kept unchanged, except for the final moves which are redirected back to the state this thread was opened from. That is, if $s_1 \xrightarrow{m} s_2$ in $\mathcal{A}^{(i,\bullet,\bullet)}_{\Gamma,z,u\vdash M}$ or $\mathcal{A}^{(i,\bullet,k)}_{\Gamma,z,v\vdash N}$ where s_2 is not final then we have transitions $(s_1, s) \xrightarrow{m} (s_2, s)$ for all $s \in \mathcal{S}$. If $s_1 \xrightarrow{j} s_2$ in $\mathcal{A}^{(i,\bullet,\bullet)}_{\Gamma,z,u\vdash M}$ where s_2 is final, then $(s_1, s) \xrightarrow{j} s$ for all $s \in \mathcal{S}$. Similarly, if $s_1 \xrightarrow{\bullet} s_2$ in $\mathcal{A}^{(i,\bullet,k)}_{\Gamma,z,v\vdash N}$ for final s_2 then we have $(s_1, s) \xrightarrow{\mathsf{ok}} s$.

- **let** $x = z(\lambda y.M)$ **in** N

  This is the final and most complicated part of the construction. Note that we must have $x : \theta_3$, $y : \theta_1$, $M : \theta_0$, $z : (\theta_1 \to \theta_0) \to \theta_3$ and $N : \theta_2$ where each θ_i is a type from Θ_i. The relevant prearena is shown in Figure 3.8 ($n \geq 0$). Play proceeds as follows:

  - After the initial move, P will play $\bullet_z$. At this point, O can either play j (an initial $[\![\theta_3]\!]$-move) or play q_0, opening an $[\![M]\!]$-thread.
  - If O chooses the latter, control shifts to the $\lambda y.M$ subterm. In a similar manner to the construction for λ-abstraction this can result in stacked copies of plays from $[\![M]\!]$. O can open a new $[\![M]\!]$-thread whenever P plays in $[\![\theta_1 \to \theta_0]\!]$ (that is either P plays a_0, closing the $[\![M]\!]$-thread, or some q_i). Note that if O opens a new $[\![M]\!]$-thread while the old one is still

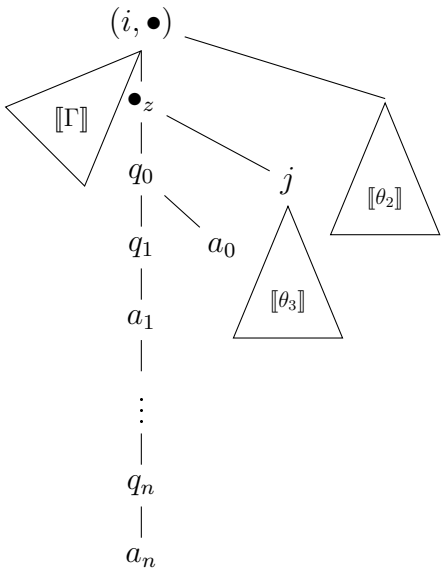


Figure 3.8: Prearena for $[\![\Gamma, (\theta_1 \to \theta_0) \to \theta_3 \vdash \theta_2]\!]$

open, the old thread will be left in a position where the only valid move is for O to answer the pending q_i with a_i. Thus, bracketing ensures that we cannot revisit an old $[\![M]\!]$-thread until we have closed the current one.

– If eventually all $[\![M]\!]$-threads are closed and O plays j, then play proceeds as in $[\![N]\!]$. In a similar way to the other $\mathbf{let}\, x = zM\, \mathbf{in}\, N$ cases all x-moves will be relabelled as z-moves.

– While playing as $[\![N]\!]$, if P ever plays in x (that is in $[\![\theta_3]\!]$), then O again gets the chance to play q_0 and open an $[\![M]\!]$-thread. If this happens then the threads can be stacked in a similar manner as before. However, an additional complication is that whenever O could open a new $[\![M]\!]$-thread, O also has the option of resuming play in $[\![N]\!]$. The resumption will have to obey bracketing, so if there are open $[\![M]\!]$-threads then for O to rejoin $[\![N]\!]$ he must play a $[\![\theta_3]\!]$-question. As before, the $[\![M]\!]$-threads interrupted in this manner can only be resumed with an O-answer, so to obey bracketing they cannot be resumed until the $[\![\theta_3]\!]$-question has been answered.

We describe below how to construct an automaton $\mathcal{A}^{(i,\bullet)}_{\Gamma,z\vdash \mathbf{let}\, x=z(\lambda y.M)\, \mathbf{in}\, N}$ for this strategy from the families of automata $\mathcal{A}^{(i,\bullet,q_0)}_{\Gamma,z,y\vdash M}$ and $\mathcal{A}^{(i,\bullet,j)}_{\Gamma,z,x\vdash N}$. We will use the stack to keep track of the stacked $[\![M]\!]$-threads and the pending question, so that we can tell whether O can resume an earlier thread without violating bracketing. We must also encode the P-pointers of moves justified by initial

x-moves (as in the other $\mathbf{let}\, x = zM\, \mathbf{in}\, N$ cases) and moves justified by q_0 (as in the λ-abstraction case).

We will need as our set of states:

- Two fresh states (1) and (2).
- Two copies of every state of each $\mathcal{A}^{(i,\bullet,j)}_{\Gamma,z,x\vdash N}$. We tag the second copy of state s as $\overset{\bullet}{s}$ to indicate that the initial x-move which opened play in $[\![N]\!]$ was a target-move.
- In a similar manner to the previous case, we define the set of states $\mathcal{S}$ (from which an $[\![M]\!]$-thread can be opened) as

  $$\mathcal{S} = \{\,(2)\,\} \uplus \{\, r \mid (r = s \vee r = \overset{\bullet}{s}) \wedge t \overset{m_x}{\to} s \text{ in some } \mathcal{A}^{(i,\bullet,j)}_{\Gamma,z,x\vdash N} \text{with } m_x \text{ a P-}x\text{-move} \,\}.$$

  For each state s from some $\mathcal{A}^{(i,\bullet,q_0)}_{\Gamma,z,y\vdash M}$ and t from $\mathcal{S}$ we will have two states (s,t) and $(\overset{\bullet}{s},t)$. These will be for when we are in an $[\![M]\!]$-thread and we need to keep track of where we are in the thread, where we should return when it is closed (or interrupted) and whether it was opened with a target-move.

The initial state will be (1) and the final states will be those of $\mathcal{A}^{(i,\bullet,j)}_{\Gamma,z,x\vdash N}$.

Our set of stack symbols will be the disjoint union of all the symbols used by each $\mathcal{A}^{(i,\bullet,j)}_{\Gamma,z,x\vdash N}$ and $\mathcal{A}^{(i,\bullet,q_0)}_{\Gamma,z,y\vdash M}$, plus the fresh symbol (1) and two copies of the states of each $\mathcal{A}^{(i,\bullet,q_0)}_{\Gamma,z,y\vdash M}$.

Our transitions will be as follows.

- $(1) \overset{\bullet/(1)}{\to} (2)$.
- From state (2) we can start $[\![N]\!]$ if there are no open $[\![M]\!]$-threads. This will be the case only if the symbol (1) is at the top of the stack. So $(2) \overset{j,(1)}{\to} i^j_N$ and $(2) \overset{\bullet}{\to} \overset{j,(1)}{\to} \overset{\bullet}{i^j_N}$ where i^j_N is the initial state in $\mathcal{A}^{(i,\bullet,j)}_{\Gamma,z,x\vdash N}$.
- We can start a new $[\![M]\!]$-thread from any state in $\mathcal{S}$. So, we have $s \overset{q_0}{\to} (i^{q_0}_M, s)$, $s \overset{\bullet}{\to} \overset{q_0}{\to} (\overset{\bullet}{i^{q_0}_M}, s)$ where $i^{q_0}_M$ is the initial state in $\mathcal{A}^{(i,q_0)}_{\Gamma,y\vdash M}$ and $s \in \mathcal{S}$.
- When we close an $[\![M]\!]$-thread we should return either to state (2) or to the current state of the $[\![N]\!]$-thread. This will be stored in the second component of the state. So if $s \overset{a_0}{\to} t$ in $\mathcal{A}^{(i,\bullet,q_0)}_{\Gamma,z,y\vdash M}$, then $(s,r) \overset{a_0}{\to} r$ and $(\overset{\bullet}{s},r) \overset{a_0}{\to} r$ for all $r \in \mathcal{S}$.

– We need to be able to open a new $[\![M]\!]$-thread or possibly resume $[\![N]\!]$ whenever $[\![M]\!]$ plays in y (and we must relabel y-moves as z-moves). If $s_1 \overset{q_i^y/\gamma}{\to} s_2 \overset{a_i^y,\gamma}{\to} s_3$ in $\mathcal{A}^{(i,\bullet,q_0)}_{\Gamma,z,y\vdash M}$ (note that — aside from pointer encodings — this must be the only transition out of s_1 as it is a P-move and that the only valid response in $[\![\Gamma, z, y \vdash M]\!]$ is for O to answer), then for all $t \in \mathcal{S}$ we have transitions

  * $(s_1, t) \overset{q_i/s_1}{\to} t$ and $(\overset{\bullet}{s_1}, t) \overset{q_i/\overset{\bullet}{s_1}}{\to} t$
  * $t \overset{a_i,s_1}{\to} (s_3, t)$ and $t \overset{a_i,\overset{\bullet}{s_1}}{\to} (\overset{\bullet}{s_3}, t)$.

  Note that the q_i or a_i from the original automata may be source- or target-moves in which case this should be preserved in the new transitions.

  Further, if $i = 1$ then we need to be able to encode a pointer and so we also have transitions $(\overset{\bullet}{s_1}, t) \overset{\overset{\circ}{q_1}/\overset{\bullet}{s_1}}{\to} t$.

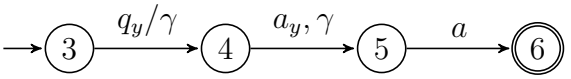
– All other transitions within each copy of $\mathcal{A}^{(i,\bullet,q_0)}_{\Gamma,z,y\vdash M}$ are preserved. If $s_1 \overset{m}{\to} s_2$ in $\mathcal{A}^{(i,\bullet,q_0)}_{\Gamma,z,y\vdash M}$ where m is a $[\![\Gamma, z]\!]$-move then $(s_1, t) \overset{m}{\to} (s_2, t)$ and $(\overset{\bullet}{s_1}, t) \overset{m}{\to} (\overset{\bullet}{s_2}, t)$ for all $t \in \mathcal{S}$.

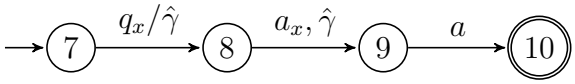
– All transitions in $\mathcal{A}^{(i,\bullet,j)}_{\Gamma,z,x\vdash N}$ are preserved but we need to rename x-moves into z-moves. So if $s_1 \overset{m}{\to} s_2$ in $\mathcal{A}^{(i,\bullet,j)}_{\Gamma,z,x\vdash N}$ where m is not an x-move then $s_1 \overset{m}{\to} s_2$ and $\overset{\bullet}{s_1} \overset{m}{\to} \overset{\bullet}{s_2}$. If $s_1 \overset{m_x}{\to} s_2$ in $\mathcal{A}^{(i,\bullet,j)}_{\Gamma,z,x\vdash N}$ where m_x is an x-move then instead we have $s_1 \overset{m_z}{\to} s_2$ and $\overset{\bullet}{s_1} \overset{m_z}{\to} \overset{\bullet}{s_2}$ where m_z is the same move but in the z-component. Additionally, if m_x is an x-move immediately justified by the initial move in $[\![\Gamma, z, x \vdash \theta_2]\!]$ then we have the transition $\overset{\bullet}{s_1} \overset{\overset{\circ}{m_z}}{\to} \overset{\bullet}{s_2}$.

Example 3.6. As an example, we apply our construction to the term

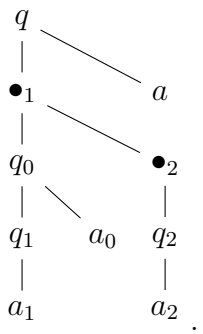
$$z : ((\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \to \mathsf{unit} \to \mathsf{unit} \vdash \mathbf{let}\, x = z(\lambda y.y())\, \mathbf{in}\, x().$$

The automata for our subterms are:

- $\mathcal{A}^{(\bullet,\bullet_y)}_{z,y\vdash y()} =$ 

- $\mathcal{A}^{(\bullet,\bullet_x)}_{z,x\vdash x()} =$ 

Combining these using our construction produces the VPA in Figure 3.9. Here we are tagging moves so that the prearena is



We can see that in states 2, 8 and $\overset{\bullet}{8}$ plays from $[\![y()]\!]$ can be nested.

This completes the inductive stage of our construction. The automata constructed actually accept a superset of the languages we are interested in. This is because during our constructions we do not attempt to restrict our automata to accept at most one source- and target-move. We do, however, ensure that each source-move can only occur if the corresponding target-move occurs earlier in the run. Due to this condition it is straightforward to restrict the automata to accept only the words we want. The final step in the construction is to restrict the languages of the automata to accept only words which either contain no source- or target-moves or have exactly one of each. This is simple to do by tripling the number of states and gives the following result:

Theorem 3.1. *For every* $\mathsf{RML}_{\text{O-Str}}$ *term* $\Gamma \vdash M : \theta$ *and every initial move* i *of the prearena* $[\![\Gamma \vdash \theta]\!]$ *we can construct a deterministic VPA* $\mathcal{A}^i_{\Gamma \vdash M}$ *which recognises the smallest set containing:*

- *For every complete play* s *in* $[\![\Gamma \vdash M : \theta]\!]_i$*, the underlying move sequence of* s*.*
- *For every complete play* s *in* $[\![\Gamma \vdash M : \theta]\!]_i$ *and every pointer from a P-question to a non-initial move in* s*, the underlying move sequence of* s *tagged with exactly one target-move and one source-move encoding that pointer.*

The family of automata $\mathcal{A}^i_{\Gamma \vdash M}$ exactly characterises $[\![\Gamma \vdash M]\!]$. Since the set of initial moves is finite and VPA equivalence is decidable, the full abstraction theorem then gives us the result below.

Theorem 3.2. *Observational equivalence of* $\mathsf{RML}_{\text{O-Str}}$*-terms is decidable.*

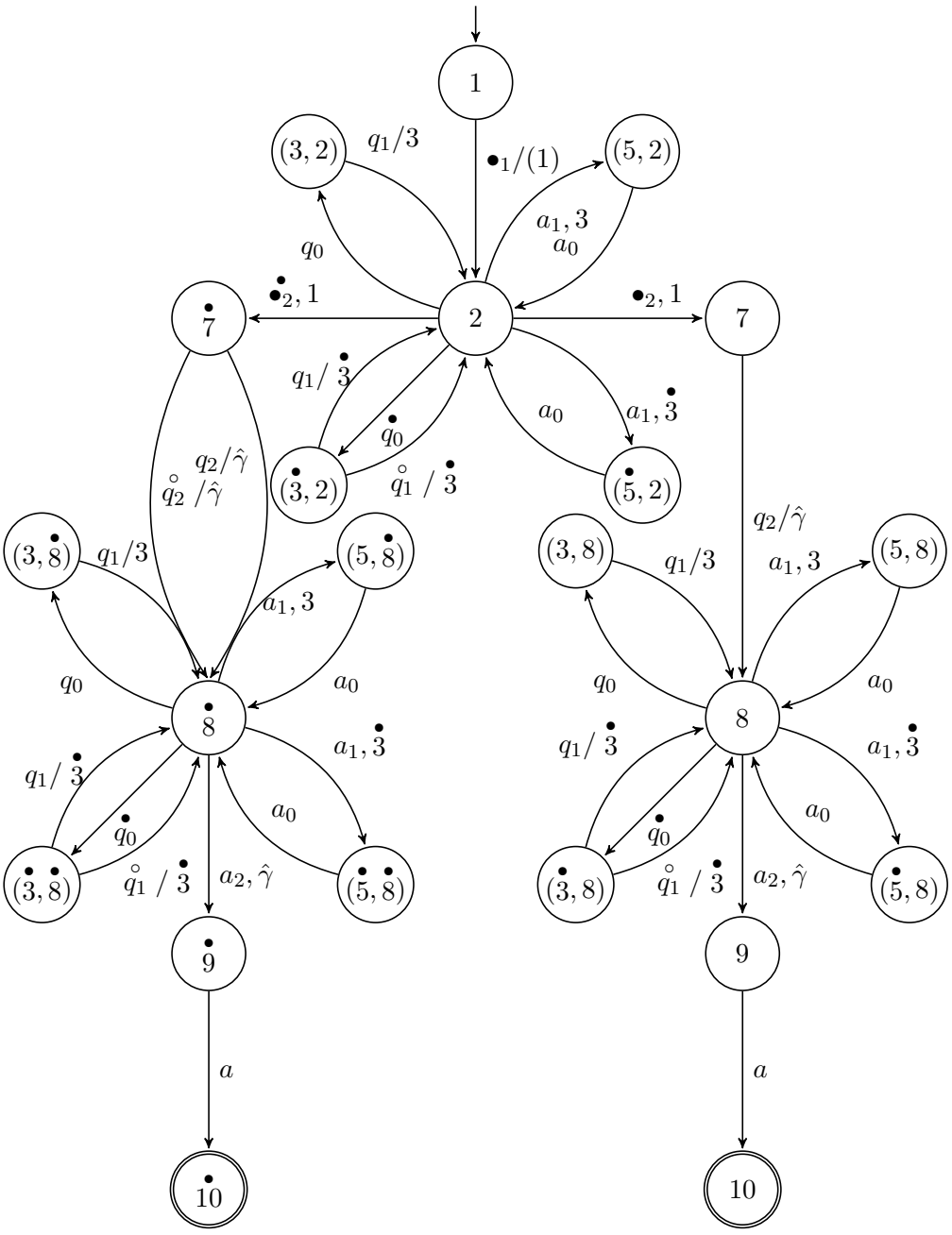


Figure 3.9: VPA for $\mathbf{let}\, x = z(\lambda y.y())\, \mathbf{in}\, x()$

3.3 Complexity

Having shown that observational equivalence is decidable for the O-strict fragment of RML, we now consider the complexity of the problem. First we consider the size of the automata our constructions produce. Following [81], we define the size of a VPA to be the sum of the number of states and the number of stack symbols. The size of the alphabet is linear in the size of the type sequent and so we ignore it. The number of transitions is bounded by a polynomial in the size of the automaton.

Proposition 3.4. *For an RML-term in canonical form $\Gamma \vdash M$ and initial $[\![\Gamma]\!]$-move i, the size of the automaton $\mathcal{A}^i_{\Gamma\vdash M}$ is bounded by an exponential in the size of $\Gamma \vdash M$.*

Proof. In each case of the construction, the set of states consists of a number of fresh states, a number of copies of the states from sub-automata and a number of copies of pairs of states from different sub-automata. The set of stack symbols is similar. The largest number of sub-automata is three (in the case of $\mathbf{let}\, x = z\, \mathbf{mkvar}(\lambda u^{\mathsf{unit}}.M, \lambda v^{\mathsf{int}}.N)\, \mathbf{in}\, Q$). This means that the size is bounded by the equation

$$|\mathcal{A}_M| \leq c \times \left(1 + \sum_{1\leq i\leq 3} |\mathcal{A}_{M_i}| + \sum_{0\leq i<j\leq 3} |\mathcal{A}_{M_i}| \times |\mathcal{A}_{M_j}|\right)$$

for some constant c where M_1,M_2,M_3 are disjoint proper subterms of M (so $|M| > |M_1|+|M_2|+|M_3|$). This gives an exponential bound on the size of the automata. □

Now we consider the total running time of the algorithm.

Proposition 3.5. *The observational equivalence problem for $\mathsf{RML}_{\text{O-Str}}$-terms in canonical form is in* ExpTime.

Proof. The time required to construct each automaton is polynomial in its size (so exponential in the size of the input). The time taken to check whether two deterministic VPA are equivalent is polynomial in the size of the two VPA. Finally, the number of VPA we will need to check is exponential in the size of the input (in the number of int-components in the context). Altogether, this gives an exponential bound on the total amount of time required to check two $\mathsf{RML}_{\text{O-Str}}$-terms in canonical form for observational equivalence. □

Note that this result only holds for $\mathsf{RML}_{\text{O-Str}}$ terms which are already in canonical form. In general, converting an RML term into canonical form can take a non-elementary amount of time. This follows from the complexity of β-reduction in the simply typed λ-calculus [102, 64]. However, in practice we would not expect to hit this bound for most terms.

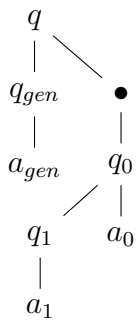


Figure 3.10: Prearena for $gen : \mathsf{unit} \to \mathsf{int} \vdash \mathbb{C} : (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$

3.3.1 Hardness

It turns out that this bound is optimal. We can show ExpTime-hardness using a reduction of the ExpTime-complete equivalence problem for nondeterministic automata on binary trees [98].

A binary-tree automaton (BTA) $\mathcal{A}$ is a tuple $\langle\, Q, \Sigma, \delta_0, \delta_2, F \,\rangle$, where Q is the finite set of states, $F \subseteq Q$ contains the final states, $\Sigma = \Sigma_0 + \Sigma_2$ is the input alphabet partitioned into the sets of nullary and binary symbols and $\delta_0 : \Sigma_0 \to 2^Q$, $\delta_2 : Q \times Q \times \Sigma_2 \to 2^Q$ are the transition functions. The automaton processes the tree bottom-up and accepts by final state. $\mathcal{T}(\mathcal{A})$ will denote the set of trees accepted by $\mathcal{A}$.

We will use plays to represent trees. A strategy is then a set of trees and we will show how to convert a BTA $\mathcal{A}$ into an $\mathsf{RML}_{\text{O-Str}}$-term whose strategy denotation represents $\mathcal{T}(\mathcal{A})$. We will do this using the type sequent $gen : \mathsf{unit} \to \mathsf{int} \vdash \mathbb{C} : (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$. The associated prearena A is shown in Figure 3.10. For simplicity, but w.l.o.g., we assume that $\Sigma_0 + \Sigma_2 = \mathsf{int}$. In fact the argument can still be carried out if Σ is larger than int by encoding each label using multiple ints.

We will represent a tree T using the play $q \bullet \mathcal{S}(T)$ on A, where $\mathcal{S}(T)$ is defined as

$$\mathcal{S}(l) := q_0\, q_{gen}\, l_{gen}\, a_0 \qquad \mathcal{S}(n(T_1, T_2)) := q_0\, q_{gen}\, n_{gen}\, q_1\, \mathcal{S}(T_1)\, a_1\, q_1\, \mathcal{S}(T_2)\, a_1\, a_0.$$

Note that $\mathcal{S}(T)$ can be seen as a record of a depth-first traversal of T. For a given tree automaton $\mathcal{A}$ we construct a term $gen : \mathsf{unit} \to \mathsf{int} \vdash \mathbb{C}_{\mathcal{A}} : (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$ such that $\mathbf{comp}(\llbracket gen \vdash \mathbb{C}_{\mathcal{A}} \rrbracket) = \{\, q \bullet \,\} \cup \{\, q \bullet \mathcal{S}(T) \mid T \in \mathcal{T}(\mathcal{A}) \,\}$. This term is given in Figure 3.11. The automata we want to simulate are nondeterministic. In order to keep track of all possible states they can reach at any stage of the computation, we rely on vectors of boolean variables $\overline{X} = \langle\, X_q \,\rangle_{q \in Q}$ (X_q is set to *true* iff we want to

represent the fact that the automaton can reach state q). To model the behaviour of the automaton at binary nodes, the sets of states reachable after processing its left subtree are recorded in $\overline{X_l}$, while those corresponding to the right one are stored in $\overline{X_r}$. δ_2 is then applied to compute the next set of states $\overline{RESULT}$, reachable after processing the binary node.

The *MODE* variable forces the players to explore only scenarios corresponding to binary trees. In particular, when the symbol acquired from *gen* is a binary label, two calls to $f()$ occur to generate the requisite two $q_1 \dots a_1$ segments.

A complete play (different from $q\bullet$) will be reached by the players only if the first q_0-thread (distinguished by its local variable *FIRST* being equal to *true*) records reachability of a final state ($\overline{RESULT} \cap F \neq \emptyset$) by $\mathcal{A}$.

Clearly, the reduction can be carried out in polynomial time, so the ExpTime-hardness of BTA equivalence implies ExpTime-hardness of observational equivalence.

Theorem 3.3. *Observational equivalence of* $\mathsf{RML}_{\text{O-Str}}$*-terms in canonical form is* ExpTime*-complete.*

Our previous proof showed that the ExpTime-bound can be reached at the type sequent

$$\mathsf{unit} \to \mathsf{int} \vdash (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}.$$

ExpTime-hardness of contextual equivalence can also be shown for $\mathsf{RML}_{\text{O-Str}}$-terms typable as

$$gen : \mathsf{unit} \to \mathsf{int}, y : ((\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \to \mathsf{unit} \vdash \mathbb{C} : \mathsf{unit}$$

The relevant prearena is shown in Figure 3.12 and the corresponding term $\mathbb{C}'_{\mathcal{A}}$ in Figure 3.13. Trees are then represented by plays $q_0 q_1 \mathcal{S}(T) a_1 a_0$, where $\mathcal{S}(T)$ is given below.

$$\mathcal{S}(l) = q_2\, q_{gen}\, l_{gen}\, a_2 \qquad \mathcal{S}(n(T_1, T_2)) = q_2\, q_{gen}\, n_{gen}\, q_3\, \mathcal{S}(T_1)\, a_3\, q_3\, \mathcal{S}(T_2)\, a_3\, a_2$$

3.4 Summary

In this chapter we have introduced the O-strict fragment of RML. This is the largest fragment of RML for which, in the game-semantic model, the location of justification pointers from O-moves is uniquely determined. This fragment consists of terms of

$$
\begin{array}{l}
\textbf{let } \overline{RESULT} = \overline{\textbf{ref } 0} \textbf{ in} \\
\textbf{let } MODE = \textbf{ref } down \textbf{ in} \\
\textbf{let } FIRST\_OPEN = \textbf{ref } false \textbf{ in} \\
\\
\lambda f^{\mathsf{com}\to\mathsf{com}}. \\
\qquad \textbf{let } FIRST = \textbf{ref } false \textbf{ in} \\
\qquad \textbf{let } \overline{X_l} = \overline{\textbf{ref } 0} \textbf{ in} \\
\qquad \textbf{let } \overline{X_r} = \overline{\textbf{ref } 0} \textbf{ in} \\
\qquad \textbf{let } Z = \textbf{ref } 0 \textbf{ in} \\
\\
\qquad \mathsf{if}\ (!FIRST\_OPEN)\ \mathsf{then}\ FIRST := false \\
\qquad \mathsf{else}\ (FIRST := true; FIRST\_OPEN := true); \\
\\
\qquad [!MODE = down]; \quad Z := gen(); \\
\\
\qquad \mathsf{if}\ (!Z \in \Sigma_0)\ \mathsf{then}\ (\overline{RESULT} := \delta_0(!Z); MODE := up); \\
\\
\qquad \mathsf{if}\ (!Z \in \Sigma_2)\ \mathsf{then} \\
\qquad\quad (\ \ f(); [!MODE = up]; \overline{X_l} := !\overline{RESULT}; \\
\qquad\quad\ \ MODE := down; \\
\qquad\quad\ \ f(); [!MODE = up]; \overline{X_r} := !\overline{RESULT}; \\
\qquad\quad\ \ \overline{RESULT} := \delta_2(!\overline{X_l}, !\overline{X_r}, !Z); MODE := up \\
\qquad\quad)\ \ ; \\
\\
\qquad [FIRST \Rightarrow (\overline{RESULT} \cap F \neq \emptyset)]
\end{array}
$$

We write $[assertion]$ as an abbreviation for $\mathsf{if}\ assertion\ \mathsf{then}\ ()\ \mathsf{else}\ \Omega$.

Figure 3.11: $gen : \mathsf{com} \to \mathsf{exp} \vdash \mathbb{C}_{\mathcal{A}} : (\mathsf{com} \to \mathsf{com}) \to \mathsf{com}$

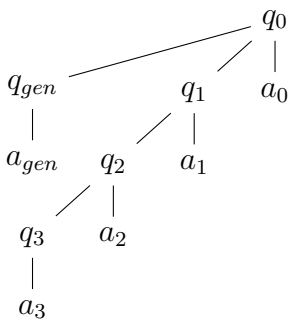


Figure 3.12: Prearena for $gen : \mathsf{unit} \to \mathsf{int}, y : ((\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \to \mathsf{unit} \vdash \mathbb{C} : \mathsf{unit}$

$$
\begin{array}{l}
\textbf{let } \overline{RESULT} = \overline{\textbf{ref } 0} \textbf{ in} \\
\textbf{let } MODE = \textbf{ref } down \textbf{ in} \\
\\
y(\lambda f^{\mathsf{unit}\to\mathsf{unit}}. \\
\qquad \textbf{let } \overline{X_l} = \overline{\textbf{ref } 0} \textbf{ in} \\
\qquad \textbf{let } \overline{X_r} = \overline{\textbf{ref } 0} \textbf{ in} \\
\qquad \textbf{let } Z = \textbf{ref } 0 \textbf{ in} \\
\\
\qquad [!MODE = down]; \quad Z := gen(); \\
\\
\qquad \mathsf{if}\ (!Z \in \Sigma_0)\ \mathsf{then}\ (\overline{RESULT} := \delta_0(!Z); MODE := up); \\
\\
\qquad \mathsf{if}\ (!Z \in \Sigma_2)\ \mathsf{then} \\
\qquad\quad (\ \ f(); [!MODE = up]; \overline{X_l} := !\overline{RESULT}; \\
\qquad\qquad MODE := down; \\
\qquad\qquad f(); [!MODE = up]; \overline{X_r} := !\overline{RESULT}; \\
\qquad\qquad \overline{RESULT} := \delta_2(!\overline{X_l}, !\overline{X_r}, !Z); MODE := up \\
\qquad\quad)\ \ ; \\
\\
\quad); [\overline{RESULT} \cap F \neq \emptyset]
\end{array}
$$

Figure 3.13: $gen : \mathsf{unit} \to \mathsf{int}, y : ((\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \to \mathsf{unit} \vdash \mathbb{C}'_{\mathcal{A}} : \mathsf{unit}$

short type (order at most two, arity at most one) which can contain free identifiers whose argument types are also short. The fragment includes complex higher-order types and examples from the literature which are known to be hard to reason about. We showed that observational equivalence for terms in this fragment is decidable. We did this by constructing (inductively over the canonical forms of the language) VPA which recognise the strategy denotation of terms. Further, we went on to show that (for terms in canonical form) the problem is ExpTime-complete.

In the next chapter we consider extensions to this fragment. We examine the effect of adding recursion (instead of only allowing **while**-loops) and of extending the types allowed. In most cases these extensions will lead to undecidability but we find a few cases where decidability can be preserved.

Chapter 4

Extensions to the O-Strict Fragment

In the previous chapter we considered the O-strict fragment of RML. We believe this is the largest fragment for which the game semantics can be represented using VPA. In this chapter we consider fragments which do not seem to be representable using VPA and consider whether observational equivalence is still decidable. We start out by considering recursion. Our previous result referred to RML with **while**-loops but no recursive functions. We show that the game semantics of recursive functions of type $\beta \to \beta$ (where $\beta \in \{\mathsf{unit}, \mathsf{int}\}$) can be represented using DPDA and hence observational equivalence remains decidable. We also show that DPDA really are required as we show hardness by representing any DPDA in this fragment of RML.

Next we consider going beyond O-strict types. The simplest such type (on the right-hand side of the turnstile) is $\beta \to \beta \to \beta$. The strategy denotations of terms of this type have plays in which O-moves have unboundedly many potential justifiers. The single-pointer representation we used to represent P-pointers does not suffice to model O-pointers. It appears that automata with infinite alphabets are needed to precisely capture such strategies. One such form of automata are Class Memory Automata. We show how deterministic CMA can be used to model terms of type $\beta \to \beta \to \beta$ and decide observational equivalence.

Despite these results, most extensions to the O-strict fragment lead to undecidability. In the final section of this chapter we prove undecidability at various type sequents and also when recursive functions of type $(\beta \to \beta) \to \beta$ are allowed.

The results presented in this chapter are joint work with Andrzej Murawski.

4.1 Recursion

The results of Chapter 3 apply to recursion-free RML (although **while**-loops are allowed). In this section we consider the effect of allowing a limited form of recursion

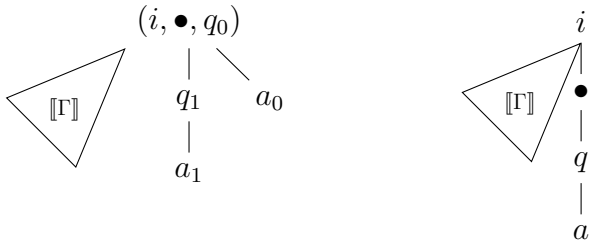


Figure 4.1: Prearenas for $[\![\Gamma, F : \beta \to \beta', x : \beta \vdash \beta']\!]$ and $[\![\Gamma \vdash \beta \to \beta']\!]$

into our language.

It is standard in call-by-value languages to only allow recursive functions which are λ-abstractions [38, 90, 108]. The simplest allowable recursive definitions will then have the form $\Gamma \vdash (\mu F : \beta \to \beta'.\lambda x : \beta.M) : \beta \to \beta'$ where $\beta, \beta' \in \{ \mathsf{unit}, \mathsf{int} \}$.

The prearena for $[\![\Gamma, F : \beta \to \beta', x : \beta \vdash \beta']\!]$ and $[\![\Gamma \vdash \beta \to \beta']\!]$ are shown in Figure 4.1. As our recursive term involves various identifiers and terms with the same type and values being passed around between them, we have various moves which appear in different components of the prearena. We will use subscripts to denote which component a move comes from and primes to denote whether the values of the moves are (or may be) different. For example, q_0 and q_1 should be taken to refer to the same move in two differing components of the prearena. However, a_1 and a_1' are moves from the same component of the prearena but are not guaranteed to have the same value.

Play in $[\![\Gamma \vdash \mu F.\lambda x.M]\!]$ proceeds as follows. P responds to the initial move, i, with $\bullet$. Then, whenever O plays q play proceeds as $[\![\Gamma, F, x \vdash M]\!]$ would respond to the initial move $(i, \bullet, q_0)$. If $[\![\Gamma, F, x \vdash M]\!]$ would answer the initial question with a_0 then instead P plays a. However, if $[\![\Gamma, F, x \vdash M]\!]$ ever questions F with q_1' then instead we start up a new thread of $[\![\Gamma, F, x \vdash M]\!]$ on initial move $(i, \bullet, q_0')$. We can open further threads in the same manner whenever P wishes to play q_1''. If P plays a_0 in any thread other than the first, this closes the current thread and play resumes in the previous thread as if a_1 had been played. This is the only way to revisit a previously open thread. In summary, $[\![\Gamma \vdash \mu F.\lambda x.M]\!]$ will consist of nested threads of $[\![\Gamma, F, x \vdash M]\!]$, with complete plays of $[\![\Gamma, F, x \vdash M]\!]$ replacing each $q_1 a_1$ pair.

It should be clear that to recognise plays from $[\![\Gamma \vdash \mu F.\lambda x.M]\!]$ we will need to be able to keep track of how many threads have been opened (that is, how many recursive calls have been made). As the depth of the recursion can be independent of the length of a play, VPA do not seem to be sufficiently expressive to capture this (the stack on a VPA can never be larger than the number of letters of the input word that have

been consumed). To capture the recursive behaviour using VPA we would need to add extra (visible) moves corresponding to calls and returns. However, functions with very different recursive behaviours can still be observationally equivalent. Making recursive calls visible would only allow us to prove equivalences where the two programs made the same recursive calls. This is clearly not what we want. In fact, the situation is very similar to adding ground recursion to IA [76]. We will follow their approach and capture strategies using DPDA. Since DPDA equivalence is decidable [99, 103] this is sufficient to show decidability of observational equivalence.

4.1.1 Representation

In a similar manner to [76] we will partition our automata into O-states and P-states according to whose turn it is to play and only P states will be allowed to perform ϵ-transitions. However, differing from their definitions, we will allow stack actions on non-ϵ-transitions so that we can reuse our existing constructions.

- In O-states the automata will be able to read an O-move from the input. If it is an answer then it will also pop a single symbol off the stack, otherwise it will leave the stack unchanged. It will then transition into a P-state.
- In a P-state the automata may be able to perform an ϵ-transition (possibly changing the stack) and transition into another P-state. The automata must be deterministic, so if a state has an outgoing ϵ-transition then either this is the only transition out of that state, or all transitions out of that state are ϵ-pop-transitions on different stack symbols. Alternatively, a P-state may be able to perform a visible transition on a P-move and transition into an O-state. If it is a question then this will push a single symbol onto the stack, otherwise it will leave the stack unchanged. There can be at most two such visible transitions from a P-state. If there are two, then they must both be on the same move, have the same destination, push the same symbol and differ only in that one of them is tagged as a source-move for encoding a P-pointer.

In our VPA construction for $\mathsf{RML}_{\text{O-Str}}$ the contents of the stack could be mapped to the sequence of open P-questions. We will insist our DPDA respect a slightly weaker condition:

- The set of open P-questions must appear in the stack.

- Immediately after a visible (i.e. non-ϵ) P-transition, the stack will be the same as it was immediately after the pending (P-)question was played (or empty if there is no such question). This implies that when in an O-state the pending question will be at the top of the stack and that when we reach a final state the stack is empty. Furthermore, we can rely on our automata to never attempt to pop the empty stack. During our constructions we often want to combine strategies and insert complete plays from one into the middle of plays from another. These properties allow us to do this safely as they ensure that if we combine automata and insert an accepting run of one DPDA into the run of another, the inserted automaton will pop exactly the symbols that it pushed, leaving the stack unchanged at the end of its run.

4.1.2 Constructions

Again we will construct our automata inductively over the canonical forms of the language. The addition of recursion requires us to add one new canonical form.

Definition 4.1. The canonical forms of RML with first-order arity one recursive functions are those of RML plus the additional form $\mathbf{let}\, y = (\mu F : \beta \to \beta'.\lambda x : \beta.\mathbb{C})z\, \mathbf{in}\, \mathbb{C}$.

Proposition 4.1. *For every term $\Gamma \vdash M : \theta$ of RML with first-order arity one recursive functions, there is a term $\Gamma \vdash N : \theta$ in canonical form, effectively constructible from M, such that $[\![\Gamma \vdash M]\!] = [\![\Gamma \vdash N]\!]$.*

Proof. The proof is almost identical to the proof of Proposition 3.3. Lemma 3.1 and Lemma 3.2 still hold and their proofs do not require any modification. The proof of Lemma 3.2 handles all the $\mathbf{let} \ldots \mathbf{in}\, \mathbb{C}$ forms in the same manner and so our new canonical form does not cause any problems. For the same reason, the addition of recursion does not break any of the cases of the inductive proof of Proposition 3.3. Hence, we just need to add a clause for terms of the form $\Gamma \vdash \mu F.\lambda x.M$. By the inductive hypothesis $\Gamma, F, x \vdash M$ can be converted to a canonical form $\Gamma, F, x \vdash C$. So we can convert the term as a whole to $\lambda y.\mathbf{let}\, z = (\mu F.\lambda x.C)y\, \mathbf{in}\, z$.

□

Now we can construct our automata. For the old canonical forms we can reuse all our existing constructions for $\mathsf{RML}_{\text{O-Str}}$, (interpreting the result as DPDA rather than VPA). There is a danger that the potential for additional stack actions on ϵ-transitions could cause our previous constructions to fail, but the properties we insist our DPDA

have ensure that this cannot happen. In the majority of cases where automata for subterms are manipulated, we only use complete plays from the strategies for subterms and these must occur in a well-nested manner. As we have already described, our automata will pop exactly the symbols they push and so when nesting complete plays in a well-bracketed manner the additional symbols that may now appear on the stack cannot break our previous constructions.

The only case in which plays are not combined in a well-bracketed manner (at least in the sense that we may return to a play coming from one subterm before the play originating in a different subterm has completed) is $[\![\Gamma \vdash \mathbf{let}\, x = z(\lambda y.M)\, \mathbf{in}\, N]\!]$. In this construction, the well-nested $[\![M]\!]$-threads can be interleaved with part of the (single) global $[\![N]\!]$-thread. However, such interruptions are started with an O-question and ended with the corresponding P-answer and so we know they leave the stack unchanged. This ensures that the construction is safe. Furthermore, all our existing constructions satisfy the required properties and so we can reuse all of our $\mathsf{RML}_{\text{O-Str}}$ constructions without modification.

For the new form, play in $[\![\Gamma, z : \beta \vdash \mathbf{let}\, y = (\mu F : \beta \to \beta'.\lambda x : \beta.M)z\, \mathbf{in}\, N]\!]$ is similar to the strategy $[\![\mu F : \beta \to \beta'.\lambda x : \beta.M]\!]$ described previously. Instead of the initial $\bullet$ followed by O playing q, we immediately start the first $[\![M]\!]$-thread on the initial move provided in z. Further $[\![M]\!]$-threads can be nested as before. If $[\![M]\!]$ wants to close the first $[\![M]\!]$-thread with a, then instead we start playing as in $[\![N]\!]$ given a as the initial y-move.

To construct $\mathcal{A}^{(i,q_z)}_{\Gamma,z\vdash \mathbf{let}\, y=(\mu F.\lambda x.M)z\, \mathbf{in}\, N}$ we take a copy of each $\mathcal{A}^{(i,q_z,\bullet,q'_x)}_{\Gamma,z,F,x\vdash M}$ for each $q' \in M_{[\![\beta]\!]}$ (so the initial moves q_z and q'_x for z and x can have different values) and an additional (marked) copy of $\mathcal{A}^{(i,q_z,\bullet,q_x)}_{\Gamma,z,F,x\vdash M}$ (where the value of the initial x-move is the same as the value of the initial z-move) to correspond to the initial call to the recursive function. To make the disjoint union over the moves of $[\![\beta]\!]$ explicit we refer to state s from $\mathcal{A}^{(i,q_z,\bullet,q'_x)}_{\Gamma,z,F,x\vdash M}$ as (s, q') and mark the additional copy $\widehat{(s,q)}$. We also need a copy of $\mathcal{A}^{(i,q_z,a_y)}_{\Gamma,z,y\vdash N}$ for each $a \in M_{[\![\beta']\!]}$. As any particular run will only involve states from one of these automata, we do not make the disjoint union as explicit for $[\![N]\!]$-states as we do for $[\![M]\!]$-states.

Our initial state will be that of the marked copy of $\mathcal{A}^{(i,q_z,\bullet,q_x)}_{\Gamma,z,F,x\vdash M}$ and the final states will be those of $\mathcal{A}^{(i,q_z,a_y)}_{\Gamma,z,y\vdash N}$. As stack symbols we use those of each $\mathcal{A}^{(i,q_z,\bullet,q'_x)}_{\Gamma,z,F,x\vdash M}$ and $\mathcal{A}^{(i,q_z,a_y)}_{\Gamma,z,y\vdash N}$ plus the states of the new automaton. Then we have the following transitions.

- If $s \overset{q''_1/\gamma}{\to} t$ in $\mathcal{A}^{(i,q_z,\bullet,q'_x)}_{\Gamma,z,F,x\vdash M}$ then $(s, q') \overset{\epsilon/(s,q')}{\to} (s_0, q'')$ where s_0 is the initial state of $\mathcal{A}^{(i,q_z,\bullet,q''_x)}_{\Gamma,z,F,x\vdash M}$.

If $q = q'$ then we also include a similar transition for the initial $[\![M]\!]$-thread, $\widehat{(s,q)} \stackrel{\epsilon/\widehat{(s,q)}}{\to} (s_0, q'')$.

These transitions correspond to recursive calls. The call is hidden and the stack is used to remember where the call was made from.

- If $s_1 \stackrel{q_1''/\gamma}{\to} s_2 \stackrel{a_1,\gamma}{\to} s_3$ in $\mathcal{A}^{(i,q_z,\bullet,q_x')}_{\Gamma,z,F,x\vdash M}$ and $t_1 \stackrel{a_0}{\to} t_2$ in $\mathcal{A}^{(i,q_z,\bullet,q_x'')}_{\Gamma,z,F,x\vdash M}$ then $(t_1, q'') \stackrel{\epsilon,(s_1,q')}{\to} (s_3, q')$.

  As in the previous case, if $q = q'$ then we include a similar transition for the initial $[\![M]\!]$-thread, $(t_1, q'') \stackrel{\epsilon,\widehat{(s_1,q)}}{\to} \widehat{(s_3, q)}$.

  These transitions corresponds to returns from recursive calls. We should return to where the call was made from.

- If $s_1 \stackrel{a_0}{\to} s_2$ in $\mathcal{A}^{(i,q_z,\bullet,q_x)}_{\Gamma,z,F,x\vdash M}$ then $\widehat{(s_1, q)} \stackrel{\epsilon}{\to} t_0$ where t_0 is the initial state of $\mathcal{A}^{(i,q_z,a_y)}_{\Gamma,z,y\vdash N}$.

  When the initial call is answered and we have a value for y, control switches to $[\![N]\!]$.

- All other transitions from $[\![M]\!]$ and all transitions from $[\![N]\!]$ are preserved. That is, if $s_1 \stackrel{m}{\to} s_2$ is a transition in $\mathcal{A}^{(i,q_z,\bullet,q_x')}_{\Gamma,z,F,x\vdash M}$ and m is not a move from $[\![F, x \vdash \beta']\!]$ then $(s_1, q') \stackrel{m}{\to} (s_2, q')$ and if $q = q'$ then $\widehat{(s_1, q)} \stackrel{m}{\to} \widehat{(s_2, q)}$. Also if $s \stackrel{m}{\to} t$ in $\mathcal{A}^{(i,q_z,a_y)}_{\Gamma,z,y\vdash N}$ then we keep this transition unchanged. Note these transitions may involve stack actions, in which case this should be the same in the new automaton as in the old.

4.1.3 Hardness

In [76], it is shown that not only is observational equivalence of third-order IA with ground recursion decidable, but that it is at least as hard as the DPDA equivalence problem. Using thunks we can easily adapt the hardness proof from [76] to show that $\mathsf{RML}_{\text{O-Str}}$ with recursive functions of type $\beta \to \beta$ is also as hard as the DPDA equivalence problem.

Suppose we are given a DPDA $\mathcal{B}$ over alphabet Σ which accepts by empty stack (initially the stack contains Z_0) and that each transition either pushes or pops one symbol. Any DPDA can be converted into such a form. We will define a term $x : \mathsf{unit} \to \mathsf{int} \vdash M_{\mathcal{B}} : \mathsf{unit}$ which encodes the language of $\mathcal{B}$. For a language $L \subseteq \Sigma^*$ we define the strategy $\widehat{L}$ by

$$\mathbf{comp}(\widehat{L}) = \{\, \bullet_l \cdot q \cdot w_1 \cdot \ldots \cdot q \cdot w_n \cdot \bullet_r \mid w_1 \ldots w_n \in L \,\}.$$

```
1  x : unit → int ⊢
2    let Q = ref q0 in
3    let top = ref Z0 in
4    let ch = ref 0 in
5      (μz : unit → unit.λy : unit .
6        let pop = ref 0 in
7        let X = ref !top in
8        while (not !pop) do
9          if δ(!Q, ϵ, !X) = (q′, α) then
10           Q := q′;
11           if α = ϵ then
12             pop := 1
13           else //α = α0 · α1, α0 =!X
14             top := α1;
15             z();
16         else
17           ch := x();
18           if δ(!Q, !ch, !X) = (q′, α) then
19             Q := q′;
20             if α = ϵ then
21               pop := 1;
22             else //α = α0 · α1, α0 =!X
23               top := α1;
24               z()
25           else Ω)()
```

Figure 4.2: The Term $M_{\mathcal{B}}$ Representing a DPDA

The term $M_{\mathcal{B}}$ such that $[\![x \vdash M_{\mathcal{B}}]\!] = \widehat{\mathcal{L}(\mathcal{B})}$ is shown in Figure 4.2. We use the call-stack to simulate the DPDA's stack. Calls correspond to pushes and returns to pops. Before each call, the value to be pushed is stored in global variable *top*; after the call it is immediately stored in X. The current input character is read from x into *ch* (after first checking whether an ϵ-transition can be performed). The **while**-loop is used to ensure we do not return until the current symbol has been popped. The term is such that given two DPDA $\mathcal{B}$ and $\mathcal{B}'$ we have $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{B}')$ if and only if $M_{\mathcal{B}} \cong M_{\mathcal{B}'}$. Constructing $M_{\mathcal{B}}$ from $\mathcal{B}$ can be done in polynomial time, giving the following result.

Theorem 4.1. *Observational equivalence of* $\mathsf{RML}_{\text{O-Str}}$ *augmented with recursive functions of type* $\beta \to \beta'$ *is decidable. Further, it is at least as hard as the DPDA equivalence problem.*

4.2 Beyond O-Strictness

Adding recursion to $\mathsf{RML}_{\text{O-Str}}$ is a strict extension which VPA are not expressive enough to capture. However, allowing recursion does not change the fact that the fragment is O-strict. Justification pointers from O-moves are still uniquely reconstructible even in the presence of recursion. In this section we consider a fragment of RML which is not O-strict but still has a decidable observational equivalence problem.

4.2.1 O-Pointers

When representing $\mathsf{RML}_{\text{O-Str}}$, we noted in Section 3.2.1 that there can be an unbounded number of potential justifiers for each P-move. Tagging moves with indices to indicate the distance between the source and target of a justification pointer seemed to require an infinite alphabet. We got around this problem by using a single-pointer representation. Each word our automata accepted contained the encoding of the location of at most one P-pointer. Since there would be an accepted word for every P-pointer which needed to be encoded, when considering the language as a whole this representation was sufficient to uniquely determine the location of all justification pointers from P-moves. This relies on the property of the O-strict fragment that if σ is a strategy over an $\mathsf{RML}_{\text{O-Str}}$ prearena and s is a sequence of moves, then there is at most one play in σ such that its underlying move sequence is s. This ensures that if we have two words on the same sequence, both encoding different pointers then we know that both those pointers occur in the same play. When we consider prearenas outside the O-strict fragment this no longer holds. There exist strategies containing two different plays whose underlying sequences of moves are the same. This can be exploited to construct two different strategies whose single-pointer representations are the same. Consequently, the single-pointer representation is not good enough outside the O-strict fragment.

Example 4.1. Consider the strategies σ_1, σ_2 on $[\![\vdash \mathsf{unit} \to \mathsf{unit} \to \mathsf{unit}]\!]$ containing exactly all even-length prefixes of the respective complete plays given below (pointers from q_1 always point at a_0 and we omit them; pointers from answers are also omitted).

$\sigma_1 :\ q_0\ a_0\ q_1\ a_1\ q_1\ a_1\ q_2\ a_2\ q_2\ a_2$ and $q_0\ a_0\ q_1\ a_1\ q_1\ a_1\ q_2\ a_2\ q_2\ a_2$

$\sigma_2 :\ q_0\ a_0\ q_1\ a_1\ q_1\ a_1\ q_2\ a_2\ q_2\ a_2$ and $q_0\ a_0\ q_1\ a_1\ q_1\ a_1\ q_2\ a_2\ q_2\ a_2$

Note that the two strategies are clearly different, yet their single-pointer representations (as used by us to encode pointers from P-moves) adapted to O-moves would be identical. This indicates that our encoding scheme for pointers cannot be extended to pointers from O-moves.

The previous example implies that in order to go beyond the O-strict fragment, we need to be able to represent multiple pointers in a single play. Since pointers may be nested or cross each other, this seemingly requires the use of an infinite alphabet. In particular, it seems that our letters should consist of the pairing of a letter from a finite alphabet (the move) and an index potentially from an infinite domain (encoding a justification pointer). Such words are referred to as *data strings*. Various classes of automata recognising data languages have been proposed, including register automata [56], pebble automata [82] and data automata [22]. For a survey of different formalisms and results see [97].

The simplest non-O-strict type (on the right-hand side of the turnstile) is $\mathsf{unit} \to \mathsf{unit} \to \mathsf{unit}$. The non-O-strict semantics of $[\![\vdash \lambda x : \mathsf{unit}.M : \mathsf{unit} \to \mathsf{unit} \to \mathsf{unit}]\!]$ consists of interleavings of plays of $[\![x : \mathsf{unit} \vdash M : \mathsf{unit} \to \mathsf{unit}]\!]$. The plays which are interleaved must satisfy a "local" condition, namely being in $[\![x \vdash M]\!]$. They cannot, however, be interleaved in any arbitrary manner. They must be interleaved in a way which satisfies some "global" condition; they must satisfy the rules of the game and in particular must satisfy the switching condition implied by visibility. The requirement that a set of sequences satisfying a local condition are interleaved in a manner respecting a global condition is very similar to the properties of languages accepted by data automata. Data automata work by first transducing the input string and checking it has the desired shape, then splitting the result into *classes* and verifying that each class is accepted by a finite automaton. However, an equivalent but simpler form of automata called *Class Memory Automata* was introduced in [21]. In the following section we review their definition and properties.

4.2.2 Class Memory Automata

We have said that to encode plays from outside the O-strict fragment we expect that the letters we use to represent moves will consist of a pair of the move itself and an index from an infinite set encoding pointer information. Formally, let Σ be a finite alphabet and Δ an infinite set. A *data word* is a finite sequence over $\Sigma \times \Delta$. We refer to values in Σ as letters and those in Δ as data values. A *data language* is a set of such words. The *string projection* of a data word is the word in Σ^* formed by taking

the first projection of each letter. For each data value d, the set of all positions in a data word w with value d is called a *class* of w. The actual data values used in our languages will be irrelevant. We will not be able to perform any operations on data values other than comparing them for equality. As such, they are only used to define classes.

Example 4.2. Let $\Sigma = \{ q_0, q_1, q_2, a_0, a_1, a_2 \}$ and $\Delta = \mathbb{N}$.

$s = (q_0, 0) \cdot (a_0, 0) \cdot (q_1, 1) \cdot (a_1, 1) \cdot (q_1, 2) \cdot (a_1, 2) \cdot (q_2, 2) \cdot (a_2, 2) \cdot (q_2, 2) \cdot (a_2, 2)$ is a data word over this alphabet. The string projection of s is $q_0 \cdot a_0 \cdot q_1 \cdot a_1 \cdot q_1 \cdot a_1 \cdot q_2 \cdot a_2 \cdot q_2 \cdot a_2$. The classes of s are

- $q_0 \cdot a_0$,
- $q_1 \cdot a_1$,
- and $q_1 \cdot a_1 \cdot q_2 \cdot a_2 \cdot q_2 \cdot a_2$.

When representing plays as data words we will insist that q_2's occur in the same class as their justifying a_1. So s could unambiguously represent the play

$$q_0 \;\; a_0 \;\; q_1 \;\; a_1 \;\; q_1 \;\; \overbrace{a_1 \;\; q_2 \;\; a_2 \;\; q_2} \;\; a_2 \; .$$

Representing strategies as data languages, we could represent σ_1 from Example 4.1 as the set of all even-prefixes of the data words

$$(q_0, i) \cdot (a_0, i) \cdot (q_1, j) \cdot (a_1, j) \cdot (q_1, k) \cdot (a_1, k) \cdot (q_2, k) \cdot (a_2, k) \cdot (q_2, k) \cdot (a_2, k)$$

$$(q_0, i) \cdot (a_0, i) \cdot (q_1, j) \cdot (a_1, j) \cdot (q_1, k) \cdot (a_1, k) \cdot (q_2, j) \cdot (a_2, j) \cdot (q_2, j) \cdot (a_2, j)$$

for any distinct i, j, k. Similarly, we can define σ_2 as the data language containing all even-prefixes of

$$(q_0, i) \cdot (a_0, i) \cdot (q_1, j) \cdot (a_1, j) \cdot (q_1, k) \cdot (a_1, k) \cdot (q_2, k) \cdot (a_2, k) \cdot (q_2, j) \cdot (a_2, j)$$

$$(q_0, i) \cdot (a_0, i) \cdot (q_1, j) \cdot (a_1, j) \cdot (q_1, k) \cdot (a_1, k) \cdot (q_2, j) \cdot (a_2, j) \cdot (q_2, k) \cdot (a_2, k)$$

for any distinct i, j, k. These two data languages will be distinct, just as the strategies they represent are inequivalent.

To recognise data languages, we will use Class Memory Automata [21].

Definition 4.2. A *Class Memory Automaton* (CMA) is a tuple $\langle Q, \Sigma, \delta, q_I, F_L, F_G \rangle$ where:

- Q is a finite set of states.
- Σ is a finite alphabet.
- $q_I \in Q$ is the initial state.
- $F_G \subseteq F_L \subseteq Q$ are the sets of globally and locally accepting states respectively.
- $\delta : (Q \times \Sigma \times (Q \cup \{\perp\})) \to \mathcal{P}(Q)$.

The transition function looks at the current state, the Σ-component of the input and the state the automaton was in last time a member of this class was read before deciding which state to move into. More formally, a *class memory function* is a function $f : \Delta \to Q \cup \{\perp\}$ such that $f(d) \neq \perp$ for only finitely many d. A configuration of a CMA is a pair (q, f) where $q \in Q$ and f is a class memory function. The initial configuration is (q_I, f_I) where $f_I(d) = \perp$ for all d. When reading a pair $(a, d) \in \Sigma \times \Delta$, the automaton can go from configuration (q, f) to (q', f') if

- $q' \in \delta(q, a, f(d))$,
- $f'(d) = q'$,
- and for all $d' \neq d$, $f'(d') = f(d')$.

The automaton accepts if, for the final configuration (q, f), $q \in F_G$ and $f(d) \in F_L \cup \{\perp\}$ for all $d \in \Delta$. A CMA is deterministic if each $\delta(p, a, q)$ is a singleton.

When representing CMA we write $q_1 \overset{(a,q_3)}{\to} q_2$ if $q_2 \in \delta(q_1, a, q_3)$, i.e. when the automaton is in state q_1, the input is (a, d) and the last state the CMA entered when reading input in class d was q_3 (or if $q_3 = \perp$ then this is the first occurrence of data value d) then the automaton can transition into state q_2.

Example 4.3. The data languages suggested for σ_1 and σ_2 in Example 4.2 are recognised by the two CMA in Figure 4.3. Those states shown as accepting are both locally and globally accepting. Note that these are both deterministic CMA and that although they appear similar they do accept different languages. The transitions out of states 9 and 13 force the data value to either be the same as the previous value or differ as appropriate.

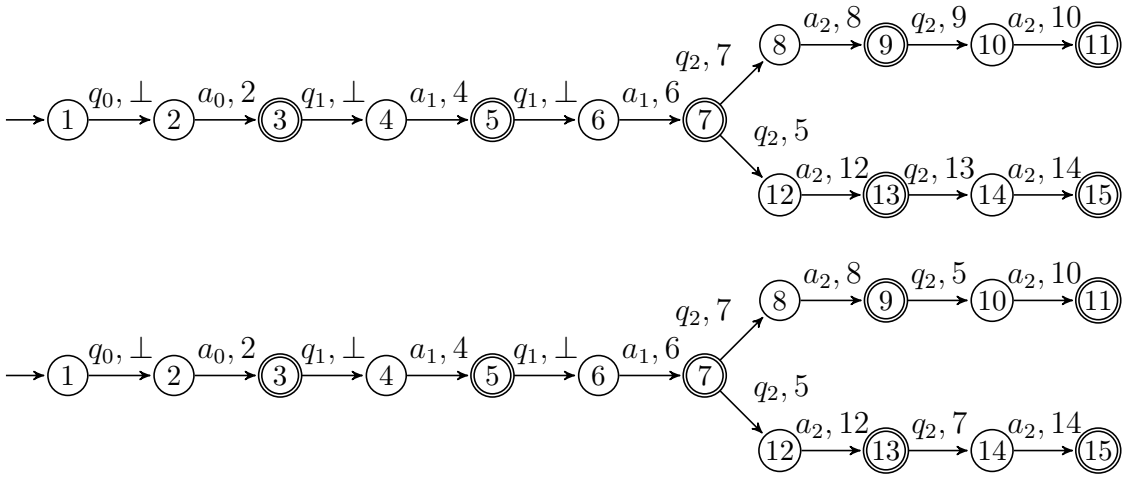


Figure 4.3: CMA for the languages in Example 4.2

4.2.2.1 Complementation

CMA are closed under intersection and have a decidable emptiness problem [21, 22]. Unfortunately, they are not closed under complementation. However, we show that for any deterministic CMA, there is a nondeterministic CMA which accepts its complement. This is sufficient for equivalence checking deterministic CMA.

Proposition 4.2. *Given a deterministic CMA* $\mathcal{C} = \langle Q, \Sigma, \delta, q_I, F_L, F_G \rangle$ *we can construct a nondeterministic CMA* $\overline{\mathcal{C}}$ *which accepts the complement of* $\mathcal{L}(\mathcal{C})$*.*

Proof. Given $\mathcal{C}$ and a data word w, $\mathcal{C}$ has exactly one run on w, ending in configuration (q, f). $\mathcal{C}$ accepts w if $q \in F_G$ and $f(d) \in F_L \cup \{ \perp \}$ for all $d \in \Delta$. Taking the contrapositive, $\mathcal{C}$ will reject if and only if either $q \notin F_G$ or there is some d appearing in w such that $f(d) \notin F_L$. We construct $\overline{\mathcal{C}}$ by guessing the reason the word is rejected. If the nondeterminism is unable to find a reason why $\mathcal{C}$ would reject the given word then it must be because $\mathcal{C}$ accepts and so we should reject.

- Our set of states is $\{ (1) \} \uplus (Q \times \{ 1, 2, 3, 4 \})$. Our four copies of Q will denote:

  1. We are guessing that the word is globally rejected.
  2. We are guessing that the word is locally rejected by some class, but have not guessed which class yet.
  3. We are guessing that the word is locally rejected by the current class.
  4. We are guessing that the word is locally rejected by a class other than this one.

- The initial state is (1). It is globally and locally accepting if and only if $\mathcal{C}$ rejects the empty word (i.e. if the initial state of $\mathcal{C}$ is globally rejecting).
- Every state $(s, 1)$ is locally accepting, but only those for which $s \notin F_G$ are globally accepting.
- Every state $(s, 2)$ is locally accepting but not globally accepting.
- Every state $(s, 3)$ is locally and globally accepting if and only if $s \notin F_L$.
- Every state $(s, 4)$ is locally and globally accepting.
- Our transitions are as follows.
  - If $q_I \overset{(m,\perp)}{\to} s_2$ then $(1) \overset{(m,\perp)}{\to} (s_2, 1)$, $(1) \overset{(m,\perp)}{\to} (s_2, 2)$ and $(1) \overset{(m,\perp)}{\to} (s_2, 3)$.
  - If $s_1 \overset{(m,s_3)}{\to} s_2$ where $s_3 \neq \perp$ then we have the following transitions:
    - $(s_1, 1) \overset{(m,(s_3,1))}{\longrightarrow} (s_2, 1)$
    - $(s_1, 2) \overset{(m,(s_3,2))}{\longrightarrow} (s_2, 2)$
    - $(s_1, 2) \overset{(m,(s_3,2))}{\longrightarrow} (s_2, 3)$
    - $(s_1, 3) \overset{(m,(s_3,3))}{\longrightarrow} (s_2, 3)$
    - $(s_1, 3) \overset{(m,(s_3,2))}{\longrightarrow} (s_2, 4)$
    - $(s_1, 3) \overset{(m,(s_3,4))}{\longrightarrow} (s_2, 4)$
    - $(s_1, 4) \overset{(m,(s_3,2))}{\longrightarrow} (s_2, 4)$
    - $(s_1, 4) \overset{(m,(s_3,3))}{\longrightarrow} (s_2, 3)$
    - $(s_1, 4) \overset{(m,(s_3,4))}{\longrightarrow} (s_2, 4)$.
  - If $s_1 \overset{m,\perp}{\longrightarrow} s_2$ then we have the following transitions:
    - $(s_1, 1) \overset{(m,\perp)}{\longrightarrow} (s_2, 1)$
    - $(s_1, 2) \overset{(m,\perp)}{\longrightarrow} (s_2, 2)$
    - $(s_1, 2) \overset{(m,\perp)}{\longrightarrow} (s_2, 3)$
    - $(s_1, 3) \overset{(m,\perp)}{\longrightarrow} (s_2, 4)$
    - $(s_1, 4) \overset{(m,\perp)}{\longrightarrow} (s_2, 4)$.

Accepting runs of $\overline{\mathcal{C}}$ can have two distinct forms, depending on whether we are guessing that the word is globally or locally rejected. If we guess the word is globally rejected then every state of the run will be in component 1. Alternatively, if we guess the word is locally rejected then the run will start with some number (possibly

zero) of transitions through states labelled 2. At this stage the automaton has not guessed which class will be locally rejecting. Eventually, the automaton transitions into a state labelled with 3. This corresponds to guessing that class will be locally rejecting. From this point on, every transition on a letter from this class will go to a state labelled with 3 and every transition on a letter from a different class will go to a state in component 4.

If word w is rejected by $\mathcal{C}$ because the unique run does not end in a globally accepting state, then the run of $\overline{\mathcal{C}}$ which initially transitions into the component labelled with 1 will accept w. If w is rejected by $\mathcal{C}$ because some class ends in a locally rejecting state, then the run which transitions into a state labelled with 3 on the first input letter of this class (and so does not move into a component labelled 1 initially) will accept. Note that the final state visited in each class on this run will always be a state labelled 2, 3 or 4 and that the final state must be labelled either 3 or 4. As such, the final state in every class will be locally accepting (the unique class in which the final state is labelled 3 is not locally accepting in $\mathcal{C}$) and the final state will be globally accepting.

Also note that if w is accepted by $\mathcal{C}$ there is no accepting run of $\overline{\mathcal{C}}$ on w. The unique run which transitions into a 1-labelled state must end in a globally rejecting state. Similarly, the unique run which always stays in 2-labelled states must end in a globally rejecting state. On any run which transitions into a state labelled with 3, the final state visited in that class will be locally rejecting. Hence, $\overline{\mathcal{C}}$ accepts exactly $\overline{\mathcal{L}(\mathcal{C})}$. □

Since CMA are closed under intersection and have a decidable emptiness problem, this shows that equivalence of deterministic CMA is decidable (as $X \subseteq Y \Leftrightarrow X \cap \overline{Y} = \emptyset$).

4.2.3 $\mathsf{RML}_{\mathrm{CMA}}$

We now consider a fragment of RML for which we can represent the game semantics using CMA. We call this fragment $\mathsf{RML}_{\mathrm{CMA}}$.

Definition 4.3. The fragment $\mathsf{RML}_{\mathrm{CMA}}$ consists of terms which can be typed

$$x_1 : \mathit{ctype}_1, \ldots, x_n : \mathit{ctype}_n \vdash M : \mathit{ttype}$$

where *ctype* and *ttype* are as follows

$$\begin{array}{rcl} \mathit{ctype} & ::= & \alpha \mid \alpha \to \mathit{ctype} \\ \alpha & ::= & \beta \mid \beta \to \beta \mid \mathsf{int\ ref} \end{array} \qquad \begin{array}{rcl} \mathit{ttype} & ::= & \alpha \mid \beta \to \alpha \\ \beta & ::= & \mathsf{unit} \mid \mathsf{int} \end{array}$$

This definition means that we allow type sequents of the form

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

For terms which can be given type α we will represent their game semantics using deterministic finite automata. We know how to do this from [73, 47]. Plays for terms which can only be given type $\beta \to \alpha$ will be represented as data words accepted by deterministic CMA. Note that the prearena will have the form in Figure 4.4. The

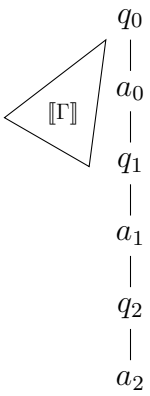


Figure 4.4: Shape of prearena for $[\![\Gamma \vdash \beta \to \alpha]\!]$

moves a_0 and a_1 must be the only move at that level (as depicted) but all the rest could potentially represent multiple moves if the corresponding part of the type is int (or $\mathsf{int\ ref}$).

The string projection of the data word will be the underlying move sequence of the play it represents, plus the encoding of at most one P-pointer in the manner we used for $\mathsf{RML}_{\text{O-Str}}$.

The data values will be such that:

- Every letter up to and including the unique occurrence of a_0 has the same data value.
- Every move hereditarily justified by a q_1 has the same data value as that q_1.
- Every $[\![\Gamma]\!]$-move played after the unique occurrence of a_0 has the same data value as the unique q_1 or q_2 which is open at that point.
- All occurrences of a_0 and q_1 have distinct data values.

This labelling scheme is sufficient to uniquely reconstruct the location of all justification pointers and is such that (the complete plays of) two strategies are equal if and only if their representations are equal. The data languages in Example 4.2 follow this encoding.

4.2.4 Construction

We will recognise strategies represented as sets of such plays using deterministic CMA. Again, during our construction we do not actually restrict our automata to having at most one P-pointer, just to be such that a source-move can only be played if the corresponding target-move was played earlier. It is then easy to restrict the final automaton to only allow plays with at most one pointer. Also as before, we will omit the initial move from plays and instead construct a family of automata for each strategy. The construction proceeds inductively over the normal forms. We only need to consider terms which have to be given type $\beta \to \alpha$. Conditionals are again trivial to handle and we consider each of the remaining normal forms we have to deal with in turn.

4.2.4.1 λ-abstraction

The strategy for $\lambda x.M$ responds to the initial move with a_0. Then, every time O plays q_1 it opens a new $[\![M]\!]$-thread. Each thread is a play from $[\![\Gamma, x : \beta \vdash M : \alpha]\!]$, only O can switch between threads and this switch can only happen after P plays a_1 or a_2. This is exactly when we have reached a complete play.

Suppose we have a family of DFA $\mathcal{M}_{(i,q_1)}$ which accepts plays of $[\![\Gamma, x \vdash M]\!]$. We will construct a family of CMA $\mathcal{C}_i$ which accepts plays from $[\![\Gamma \vdash \lambda x.M]\!]$.

- The set of states is the disjoint union of the set of states of each $\mathcal{M}_{(i,q_1)}$, plus fresh states (1) and (2).
- The initial state is (1).
- The sets of globally and locally final states are equal and consist of the final states of each $\mathcal{M}_{(i,q_1)}$ plus (2).
- Our transition relation is as follows:
  - (1) $\overset{(a_0,\perp)}{\to}$ (2).
  - For each q_1, (2) $\overset{(q_1,\perp)}{\to} I_{q_1}$ where I_{q_1} is the initial state of $\mathcal{M}_{(i,q_1)}$.

– If $s_1 \xrightarrow{m} s_2$ in $\mathcal{M}_{(i,q_1)}$ then $s_1 \xrightarrow{(m,s_1)} s_2$.

– If f is final in $\mathcal{M}_{(i,q_1)}$, then for any q_1' we have $f \xrightarrow{(q_1',\perp)} I_{q_1'}$ where $I_{q_1'}$ is the initial state of $\mathcal{M}_{(i,q_1')}$ (we use the prime to emphasise that although q_1 and q_1' come from the same part of the prearena, they may be different moves).

– If $s_1 \xrightarrow{q_2} s_2$ in $\mathcal{M}_{(i,q_1)}$ and f is final in some $\mathcal{M}_{(i,q_1')}$ then $f \xrightarrow{(q_2,s_1)} s_2$.

Note that this is deterministic. The overlap between the third and fifth clauses is not a problem as in the cases that fall in the intersection they define the same transition.

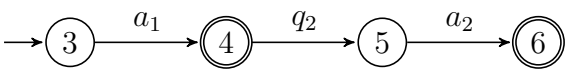
After the initial a_0, the language accepted will be an interleaving of words from $\mathcal{M}_{(i,q_1)}$ (started by q_is). Each thread can only be started from an accepting state, which can only have been reached by an a_i. Furthermore, we preserve the invariant that at most one class memory is a non-final state. Note that when $\mathcal{M}_{(i,q_1)}$ is in an accepting state, the only letters it can transition on are q_2's. Further, $\mathcal{M}_{(i,q_1)}$ can never transition on a q_2 when not in a final state. This ensures that switching can only happen after an a_i.

We do not need to add any additional encoding of pointers as there are no new P-questions.

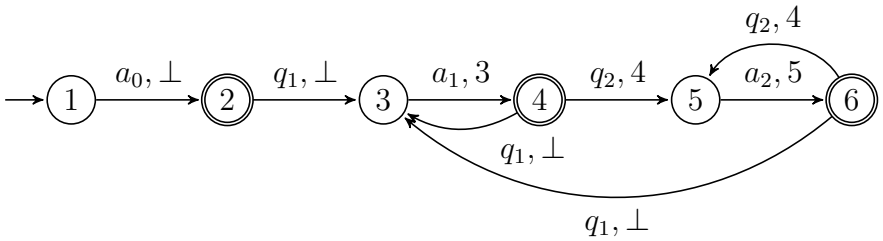
Example 4.4. As an example, let us consider the term

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

The DFA representing $[\![x \vdash \mathbf{let}\, c = \mathbf{ref}\, 0\, \mathbf{in}\, \lambda y : \mathsf{unit}.\mathbf{if}\, !c = 0\, \mathbf{then}\, c := 1\, \mathbf{else}\, \Omega]\!]$ is shown below.



Applying our construction to λ-abstract x gives the automaton below. The states shown as accepting are both locally and globally accepting. Note that after the initial a_0, the language accepted will consist of interleavings of classes each of which is either $q_1 \cdot a_1$ or $q_1 \cdot a_1 \cdot q_2 \cdot a_2$. That is, it will consist of interleavings of complete plays from the DFA for $[\![x \vdash \mathbf{let}\, c = \mathbf{ref}\, 0\, \mathbf{in}\, \lambda y : \mathsf{unit}.\mathbf{if}\, !c = 0\, \mathbf{then}\, c := 1\, \mathbf{else}\, \Omega]\!]$ (prefixed with the initial x-move q_1). Further, the switching between classes can only occur after P has played (i.e. only O can switch threads).



4.2.4.2 $\mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}\, M$

This construction is standard. We must restrict the behaviour of x to that of a good variable and then hide all x-moves. We are given a family of CMA $\mathcal{C}_{i,\bullet}$ for $[\![\Gamma, x : \mathsf{int\ ref} \vdash M : \beta \to \alpha]\!]$. We construct a family of CMA which recognise $[\![\Gamma \vdash \mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}\, M]\!]$. We perform the construction in two stages.

Restriction to good-variable behaviour First we restrict the behaviour of $\mathcal{C}_{i,\bullet}$ to good-variable behaviour by storing the value of x in the states. We refer to the resulting family of CMA as $\mathcal{D}_i$.

- The set of states is the set of pairs of a state from $\mathcal{C}_{i,\bullet}$ and an integer value from int.
- The initial state is $(I, 0)$ where I is initial in $\mathcal{C}_{i,\bullet}$. The locally (resp. globally) final states are those of the form (f, j) where f is locally (resp. globally) final in $\mathcal{C}_{i,\bullet}$.
- If $s_1 \xrightarrow{(m,s_3)} s_2$ in $\mathcal{C}_{i,\bullet}$ where m is not an x-write or the response to an x-read and $s_3 \neq \bot$ then $(s_1, j) \xrightarrow{(m,(s_3,k))} (s_2, j)$ for all j, k.
- If $s_1 \xrightarrow{(m,\bot)} s_2$ in $\mathcal{C}_{i,\bullet}$ where m is not an x-write or the response to an x-read then $(s_1, j) \xrightarrow{(m,\bot)} (s_2, j)$ for all j.
- If $s_1 \xrightarrow{(\mathsf{write}(l)_x, s_1)} s_2$ in $\mathcal{C}_{i,\bullet}$ (note that we know all x-moves must occur in the same class as the preceding move) then $(s_1, j) \xrightarrow{(\mathsf{write}(l)_x, (s_1,j))} (s_2, l)$ for all j.

  Also, in case P's first move is a an x-write, if $s_1 \xrightarrow{(\mathsf{write}(l)_x, \bot)} s_2$ (in which case s_1 must be initial) in $\mathcal{C}_{i,\bullet}$ then $(s_1, 0) \xrightarrow{(\mathsf{write}(l)_x, \bot)} (s_2, l)$.
- If $s_1 \xrightarrow{(j_x, s_1)} s_2$ in $\mathcal{C}_{i,\bullet}$ (where j_x is the response to an x-read) then $(s_1, j) \xrightarrow{(j_x, (s_1,j))} (s_2, j)$.

$\mathcal{D}_i$ will be deterministic and accept all complete plays in $\mathcal{C}_{i,\bullet}$ which obey good-variable behaviour.

Hiding We now consider how to hide the x-moves. If we are in a configuration (s_1, f) of $\mathcal{D}_i$ where we can perform a transition $s_1 \overset{(m_x, s_3)}{\to} s_2$ where m_x is an x-move then, due to the determinacy of our strategies and the restriction to good-variable behaviour, this must be the only possible transition we can perform from this configuration. Further, we know this transition must occur in the same class as the preceding move, so (unless this is P's first move) $s_3 = s_1$. Hence, for every state s_0 of $\mathcal{D}_i$ there is a unique maximal (and not necessarily finite) sequence of transitions

$$s_0 \overset{(m_0, s_0')}{\to} s_1 \overset{(m_1, s_1)}{\to} s_2 \overset{(m_2, s_2)}{\to} \dots$$

where each m_i is an x-move and either $s_0' = s_0$ or s_0 is initial in which case $s_0' = \bot$.

We construct a family of CMA $\mathcal{E}_i$ for $[\![\Gamma \vdash \mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}\, M]\!]$ by considering where this sequence ends up for each state. We keep everything the same as $\mathcal{D}_i$ except the transition relation.

- If the maximal sequence of x-moves out of state s_0 is empty then all transitions out of state s_0 are unchanged.
- If the maximal sequence out of s_0 is finite and non-empty and ends in state s_n and s_n contains a transition $s_n \overset{(m, s_n)}{\to} s_{n+1}$ (note that m cannot be an x-move and this transition must be in the same class as all the x-transitions) then we add a transition $s_0 \overset{(m, s_0')}{\to} s_{n+1}$.
- All transitions on x-moves are removed.

The resulting CMA is deterministic and accepts the language of $\mathcal{D}_i$ with all x-moves hidden. That is, it recognises $[\![\Gamma \vdash \mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}\, M]\!]$.

4.2.4.3 $\mathbf{let}\, x^\beta = N\, \mathbf{in}\, M$

The strategy for this term essentially consists of the concatenation of the strategies for $[\![N]\!]$ and $[\![M]\!]$. We are given a family of DFA $\mathcal{A}_i$ for $[\![\Gamma \vdash N : \beta]\!]$ and a family of CMA $\mathcal{C}_{i,j}$ for $[\![\Gamma, x : \beta \vdash M : \beta \to \alpha]\!]$. We assume that each $\mathcal{A}_i$ and $\mathcal{C}_{i,j}$ never return to their initial states (this should be true already for the $\mathcal{C}_{i,j}$ and is easy to ensure for $\mathcal{A}_i$).

We construct a family of CMA $\mathcal{D}_i$ for $[\![\Gamma \vdash \mathbf{let}\, x^\beta = N\, \mathbf{in}\, M]\!]$.

If $\mathcal{L}(\mathcal{A}_i) = \{\, j \,\}$ then $\mathcal{D}_i = \mathcal{C}_{i,j}$. Otherwise:

- The set of states is the disjoint union of the set of states of $\mathcal{A}_i$ and each $\mathcal{C}_{i,j}$.

- The initial state is that of $\mathcal{A}_i$.
- The locally and globally accepting states are those from each $\mathcal{C}_{i,j}$.
- If $I \stackrel{m}{\rightarrow} s$ in $\mathcal{A}_i$ where I is the initial state then $I \stackrel{(m,\perp)}{\rightarrow} s$.
- If $s_1 \stackrel{m}{\rightarrow} s_2$ in $\mathcal{A}_i$ where s_1 is not initial and s_2 is not final then $s_1 \stackrel{(m,s_1)}{\rightarrow} s_2$.
- If $s_1 \stackrel{j}{\rightarrow} s_2$ in $\mathcal{A}_i$ where s_2 is final (so s_1 is not initial) and $I_j \stackrel{(m,\perp)}{\rightarrow} s_3$ in $\mathcal{C}_{i,j}$ where I_j is the initial state of $\mathcal{C}_{i,j}$ then $s_1 \stackrel{(m,s_1)}{\rightarrow} s_3$.
- All other $\mathcal{C}_{i,j}$ transitions are preserved unchanged. That is, if $s_1 \stackrel{(m,s_3)}{\rightarrow} s_2$ in $\mathcal{C}_{i,j}$ where s_1 is not initial (and s_3 could be $\perp$) we have the same transition in $\mathcal{D}_i$.

Determinacy is inherited from $\mathcal{A}_i$ and $\mathcal{C}_{i,j}$.

4.2.4.4 $\mathbf{let}\, x = zy^{\beta}\, \mathbf{in}\, M$

We assume that x is not of type β as otherwise this can be handled in the manner described in Section 4.2.4.3.

Play starts off with P copying the component of the initial move corresponding to y into the z component with the move j_z. O must respond with the unique answer, $\bullet_z$. From then on play is exactly as in $[\![\Gamma, z, y, x \vdash M]\!]$ except that all x-moves are relabelled as z-moves. Any such move which was justified by the initial move is now justified by the occurrence of $\bullet_z$ O was forced to play. These pointers will have to be made explicit as part of our construction.

We are given CMA $\mathcal{C}_{(i,\bullet,j,\bullet_x)}$ for $[\![\Gamma, z : \beta \rightarrow \mathit{ctype}, y : \beta, x : \mathit{ctype} \vdash M : \beta \rightarrow \alpha]\!]$. We assume that each $\mathcal{C}_{(i,\bullet,j,\bullet_x)}$ can never re-enter its initial state.

We construct a family of CMA $\mathcal{D}_{(i,\bullet,j)}$ for $[\![\Gamma, z, y \vdash \mathbf{let}\, x = zy^{\beta}\, \mathbf{in}\, M]\!]$.

- As our set of states we take two copies of the set of states of $\mathcal{C}_{(i,\bullet,j,\bullet_x)}$. The second copy is required to encode P-pointers and so we tag state s in the second copy as $\overset{\bullet}{s}$. We also need two fresh states (1) and (2).
- The initial state is (1).
- The locally and globally accepting states are those of $\mathcal{C}_{(i,\bullet,j,\bullet_x)}$ (both tagged and untagged).
- The transitions are:

- (1) $\overset{(j_z,\perp)}{\to}$ (2) where j_z is the initial move for y copied into the z-component.
- (2) $\overset{(\bullet_z,(2))}{\to}$ s_0 and (2) $\overset{(\overset{\bullet}{\bullet_z},(2))}{\to}\overset{\bullet}{s_0}$ where s_0 is the initial state of $\mathcal{C}_{(i,\bullet,j,\bullet_x)}$.
- If $s_0 \overset{(m,\perp)}{\to} s_1$ in $\mathcal{C}_{(i,\bullet,j,\bullet_x)}$ where s_0 is the initial state of $\mathcal{C}_{(i,\bullet,j,\bullet_x)}$ and m is not an x-move then $s_0 \overset{(m,s_0)}{\to} s_1$ and $\overset{\bullet}{s_0} \overset{(m,\overset{\bullet}{s_0})}{\to} \overset{\bullet}{s_1}$.
- If $s_0 \overset{(m_x,\perp)}{\to} s_1$ in $\mathcal{C}_{(i,\bullet,j,\bullet_x)}$ where s_0 is the initial state of $\mathcal{C}_{(i,\bullet,j,\bullet_x)}$ and m_x is an x-move then $s_0 \overset{(m_z,s_0)}{\to} s_1$, $\overset{\bullet}{s_0} \overset{(m_z,\overset{\bullet}{s_0})}{\to} \overset{\bullet}{s_1}$ and $\overset{\bullet}{s_0} \overset{(\overset{\circ}{m_z},\overset{\bullet}{s_0})}{\to} \overset{\bullet}{s_1}$ where m_z is the relabelling of m_x into the z-component.
- If $s_1 \overset{(m,s_3)}{\to} s_2$ in $\mathcal{C}_{(i,\bullet,j,\bullet_x)}$ where s_1 is not the initial state of $\mathcal{C}_{(i,\bullet,j,\bullet_x)}$ and m is not an x-move then $s_1 \overset{(m,s_3)}{\to} s_2$ and $\overset{\bullet}{s_1} \overset{(m,\overset{\bullet}{s_3})}{\to} \overset{\bullet}{s_2}$. If $s_3 = \perp$ then instead of $\overset{\bullet}{s_3}$ the second transition should use $\perp$.
- If $s_1 \overset{(m_x,s_1)}{\to} s_2$ in $\mathcal{C}_{(i,\bullet,j,\bullet_x)}$ where m_x is an x-move then $s_1 \overset{(m_z,s_1)}{\to} s_2$ and $\overset{\bullet}{s_1} \overset{(m_z,\overset{\bullet}{s_1})}{\to} \overset{\bullet}{s_2}$ where m_z is the relabelling of m_x into the z-component. Additionally, if m_x was justified by the initial move then we also have a transition $\overset{\bullet}{s_1} \overset{(\overset{\circ}{m_z},\overset{\bullet}{s_1})}{\to} \overset{\bullet}{s_2}$.

4.2.4.5 $\mathbf{let}\, x = z(\lambda y.M)\, \mathbf{in}\, N$

Again we assume that x is not of type β as otherwise this can be handled as in Section 4.2.4.3. Our subterms must be typed $\Gamma, z : (\beta \to \beta) \to \mathit{ctype}, y : \beta \vdash M : \beta$ and $\Gamma, z : (\beta \to \beta) \to \mathit{ctype}, x : \mathit{ctype} \vdash N : \beta \to \alpha$. The prearena for this type sequent is shown in Figure 4.5. Play starts with P playing $\bullet$. O can then either play j_z, starting

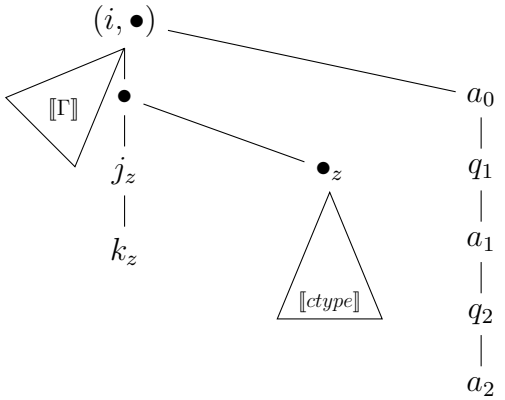


Figure 4.5: Prearena for $[\![\Gamma, z : (\beta \to \beta) \to \mathit{ctype} \vdash N : \beta \to \alpha]\!]$

an $[\![M]\!]$-thread, or play $\bullet_z$ the initial x-move. If O chooses the former, then this

$[\![M]\!]$-thread must be played to completion before anything else can happen. Once it is finished (with P playing k_z as the final move) we return to the situation where O can either play j_z or $\bullet_z$.

Once O does eventually play the initial x-move $\bullet_z$ we can start playing as $[\![N]\!]$. We play exactly as $[\![N]\!]$ except that all x-moves are renamed into z-moves. Additionally, whenever P plays in x (which becomes a z-move), O can again play j_z and start an $[\![M]\!]$-thread. Again this thread must be played to completion before we can do anything else.

We are given a family of DFA $\mathcal{M}_{(i,\bullet,j)}$ for $[\![\Gamma, z, y \vdash M]\!]$ and a family of CMA $\mathcal{N}_{(i,\bullet,\bullet_x)}$ for $[\![\Gamma, z, x \vdash N]\!]$. We again assume that none of our automata can return to their initial states. We construct a family of CMA $\mathcal{C}_{(i,\bullet)}$ which recognise $[\![\Gamma, z \vdash \mathbf{let}\, x = z(\lambda y.M)\, \mathbf{in}\, N]\!]$.

- Our set of states consists of
  - Two fresh states (1) and (2).
  - Two copies of the set of states of $\mathcal{N}_{(i,\bullet,\bullet_x)}$. We mark states of the second copy $\overset{\bullet}{s}$.
  - Define $\mathcal{S}$, the states from which an $[\![M]\!]$-thread can be opened, as

    $$\mathcal{S} = \{\,(2)\,\} \uplus \{\, r \mid (r = s \vee r = \overset{\bullet}{s}) \wedge t \overset{m_x}{\to} s \text{ in } \mathcal{N}_{(i,\bullet,\bullet_x)} \text{with } m_x \text{ a P-}x\text{-move} \,\}.$$

    We will then take states (s, t) where s is a state from some $\mathcal{M}_{(i,\bullet,j)}$ and t is a state in $\mathcal{S}$.
- The initial state is (1).
- The local and global final states are those of $\mathcal{N}_{(i,\bullet,\bullet_x)}$ (both tagged and untagged; we will never enter a final state paired with an $\mathcal{M}_{(i,\bullet,j)}$ state).
- The transitions will be:
  - $(1) \overset{(\bullet,\perp)}{\to} (2)$.
  - $(2) \overset{(\bullet_z,(2))}{\to} s_0$ and $(2) \overset{(\overset{\bullet}{\bullet}_z,(2))}{\to} \overset{\bullet}{s_0}$ where s_0 is the initial state of $\mathcal{N}_{(i,\bullet,\bullet_x)}$.
  - If $s_0 \overset{(m,\perp)}{\to} s_1$ in $\mathcal{N}_{(i,\bullet,\bullet_x)}$ where s_0 is the initial state of $\mathcal{N}_{(i,\bullet,\bullet_x)}$ and m is not an x-move then $s_0 \overset{(m,s_0)}{\to} s_1$ and $\overset{\bullet}{s_0} \overset{(m,\overset{\bullet}{s_0})}{\to} \overset{\bullet}{s_1}$.

- If $s_0 \overset{(m_x,\bot)}{\to} s_1$ in $\mathcal{N}_{(i,\bullet,\bullet_x)}$ where s_0 is the initial state of $\mathcal{N}_{(i,\bullet,\bullet_x)}$ and m_x is an x-move then $s_0 \overset{(m_z,s_0)}{\to} s_1$, $\overset{\bullet}{s}_0 \overset{(m_z,\overset{\bullet}{s}_0)}{\to} \overset{\bullet}{s}_1$ and $\overset{\bullet}{s}_0 \overset{(\overset{\circ}{m}_z,\overset{\bullet}{s}_0)}{\to} \overset{\bullet}{s}_1$ where m_z is the relabelling of m_x into the z-component.
- If $s_1 \overset{(m,s_3)}{\to} s_2$ in $\mathcal{N}_{(i,\bullet,\bullet_x)}$ where s_1 is not the initial state of $\mathcal{N}_{(i,\bullet,\bullet_x)}$ and m is not an x-move then $s_1 \overset{(m,s_3)}{\to} s_2$ and $\overset{\bullet}{s}_1 \overset{(m,\overset{\bullet}{s}_3)}{\to} \overset{\bullet}{s}_2$. If $s_3 = \bot$ then instead of $\overset{\bullet}{s}_3$ the second transition should use $\bot$.
- If $s_1 \overset{(m_x,s_1)}{\to} s_2$ in $\mathcal{N}_{(i,\bullet,\bullet_x)}$ where m_x is an x-move then $s_1 \overset{(m_z,s_1)}{\to} s_2$ and $\overset{\bullet}{s}_1 \overset{(m_z,\overset{\bullet}{s}_1)}{\to} \overset{\bullet}{s}_2$ where m_z is the relabelling of m_x in the z-component. Additionally, if m_x was justified by the initial move then we also have a transition $\overset{\bullet}{s}_1 \overset{(\overset{\circ}{m}_z,\overset{\bullet}{s}_1)}{\to} \overset{\bullet}{s}_2$.
- If $s \in \mathcal{S}$ then for all j we have $s \overset{(j_z,s)}{\to} (I_j, s)$ where I_j is the initial state of $\mathcal{M}_{(i,\bullet,j)}$.
- If $s_1 \overset{m}{\to} s_2$ in $\mathcal{M}_{(i,\bullet,j)}$ where s_2 is not final, we have $(s_1, s) \overset{(m,(s_1,s))}{\to} (s_2, s)$ where $s \in \mathcal{S}$.
- If $s_1 \overset{k}{\to} s_2$ in $\mathcal{M}_{(i,\bullet,j)}$ where s_2 is final, we have $(s_1, s) \overset{(k_z,(s_1,s))}{\to} s$ for all $s \in \mathcal{S}$ where k_z is the relabelling of k into the z-component.

4.2.4.6 $\mathbf{let}\, x = z(\mathbf{mkvar}(\lambda u.M, \lambda v.N))\, \mathbf{in}\, P$

This construction is very similar to the last case. We again assume x is not of type β. We are given families of DFA $\mathcal{M}_{(i,\bullet,\bullet)}$ for $[\![\Gamma, z, u \vdash M]\!]$ and $\mathcal{N}_{(i,\bullet,j)}$ for $[\![\Gamma, z, v \vdash N]\!]$ and a family of CMA $\mathcal{P}_{(i,\bullet,\bullet_x)}$ for $[\![\Gamma, z, x \vdash P]\!]$. We assume that none of our automata can return to their initial states. We construct a family of CMA $\mathcal{C}_{(i,\bullet)}$ for $[\![\Gamma, z \vdash \mathbf{let}\, x = z(\mathbf{mkvar}(\lambda u.M, \lambda v.N))\, \mathbf{in}\, P]\!]$.

- Our set of states consists of
  - Two fresh states (1) and (2).
  - Two copies of the set of states of $\mathcal{P}_{(i,\bullet,\bullet_x)}$. We mark states of the second copy $\overset{\bullet}{s}$.
  - Define $\mathcal{S}$, the set of states from which $[\![M]\!]$-thread and $[\![N]\!]$-threads can be opened, as

    $$\mathcal{S} = \{\,(2)\,\} \uplus \{\, r \mid (r = s \vee r = \overset{\bullet}{s}) \wedge t \overset{m_x}{\to} s \text{ in } \mathcal{P}_{(i,\bullet,\bullet_x)} \text{with } m_x \text{ a P-}x\text{-move} \,\}.$$

    We will then take states (s, t) where s is a state from some $\mathcal{M}_{(i,\bullet,\bullet)}$ or $\mathcal{N}_{(i,\bullet,j)}$ and t is a state in $\mathcal{S}$.

- The initial state is (1).
- The local and global final states are those of $\mathcal{P}_{(i,\bullet,\bullet_x)}$.
- The transitions will be:
  - $(1) \overset{(\bullet,\perp)}{\to} (2)$.
  - $(2) \overset{(\bullet_z,(2))}{\to} s_0$ and $(2) \overset{(\overset{\bullet}{\bullet}_z,(2))}{\to} \overset{\bullet}{s_0}$ where s_0 is the initial state of $\mathcal{P}_{(i,\bullet,\bullet_x)}$.
  - If $s_0 \overset{(m,\perp)}{\to} s_1$ in $\mathcal{P}_{(i,\bullet,\bullet_x)}$ where s_0 is the initial state of $\mathcal{P}_{(i,\bullet,\bullet_x)}$ and m is not an x-move then $s_0 \overset{(m,s_0)}{\to} s_1$ and $\overset{\bullet}{s_0} \overset{(m,\overset{\bullet}{s_0})}{\to} \overset{\bullet}{s_1}$.
  - If $s_0 \overset{(m_x,\perp)}{\to} s_1$ in $\mathcal{P}_{(i,\bullet,\bullet_x)}$ where s_0 is the initial state of $\mathcal{P}_{(i,\bullet,\bullet_x)}$ and m_x is an x-move then $s_0 \overset{(m_z,s_0)}{\to} s_1$, $\overset{\bullet}{s_0} \overset{(m_z,\overset{\bullet}{s_0})}{\to} \overset{\bullet}{s_1}$ and $\overset{\bullet}{s_0} \overset{(\overset{\circ}{m}_z,\overset{\bullet}{s_0})}{\to} \overset{\bullet}{s_1}$ where m_z is the relabelling of m_x into the z-component.
  - If $s_1 \overset{(m,s_3)}{\to} s_2$ in $\mathcal{P}_{(i,\bullet,\bullet_x)}$ where s_1 is not the initial state of $\mathcal{P}_{(i,\bullet,\bullet_x)}$ and m is not an x-move then $s_1 \overset{(m,s_3)}{\to} s_2$ and $\overset{\bullet}{s_1} \overset{(m,\overset{\bullet}{s_3})}{\to} \overset{\bullet}{s_2}$. If $s_3 = \perp$ then instead of $\overset{\bullet}{s_3}$ the second transition should use $\perp$.
  - If $s_1 \overset{(m_x,s_1)}{\to} s_2$ in $\mathcal{P}_{(i,\bullet,\bullet_x)}$ where m_x is an x-move then $s_1 \overset{(m_z,s_1)}{\to} s_2$ and $\overset{\bullet}{s_1} \overset{(m_z,\overset{\bullet}{s_1})}{\to} \overset{\bullet}{s_2}$ where m_z is the relabelling of m_x into the z-component. Additionally, if m_x was justified by the initial move then we also have a transition $\overset{\bullet}{s_1} \overset{(\overset{\circ}{m}_z,\overset{\bullet}{s_1})}{\to} \overset{\bullet}{s_2}$.
  - If $s \in \mathcal{S}$ then $s \overset{(\mathsf{read}_z,s)}{\to} (t_0, s)$ where t_0 is the initial state of $\mathcal{M}_{(i,\bullet,\bullet)}$. Similarly, for all $j \in \mathsf{int}$ we have $s \overset{(\mathsf{write}(j)_z,s)}{\to} (r_0^j, s)$ where r_0^j is the initial state of $\mathcal{N}_{(i,\bullet,j)}$.
  - If $s_1 \overset{m}{\to} s_2$ in $\mathcal{M}_{(i,\bullet,\bullet)}$ or $\mathcal{N}_{(i,\bullet,j)}$ where s_2 is not final, we have $(s_1, s) \overset{(m,(s_1,s))}{\to} (s_2, s)$ where $s \in \mathcal{S}$.
  - If $s_1 \overset{j}{\to} s_2$ in $\mathcal{M}_{(i,\bullet,\bullet)}$ where s_2 is final, we have $(s_1, s) \overset{(j_z,(s_1,s))}{\to} s$ for all $s \in \mathcal{S}$ where j_z is the relabelling of j into the z-component. Similarly, if $s_1 \overset{\bullet}{\to} s_2$ in $\mathcal{N}_{(i,\bullet,j)}$ where s_2 is final, we have $(s_1, s) \overset{(\mathsf{ok}_z,(s_1,s))}{\to} s$ for all $s \in \mathcal{S}$.

This completes our constructions. For any $\mathsf{RML}_{\mathrm{CMA}}$-term, the CMA constructed to recognise its game semantics will be deterministic. Hence, we get the following result.

Theorem 4.2. *Observational equivalence of* $\mathsf{RML}_{\mathrm{CMA}}$ *is decidable.*

4.3 Undecidability

We have shown that observational equivalence of the two fragments of RML $\mathsf{RML}_{\text{O-Str}}$ and $\mathsf{RML}_{\text{CMA}}$ are decidable. These fragments are not directly comparable; each contains type sequents the other does not. So, Theorem 4.2 does not strictly extend the result of Theorem 3.2. It does, though, add to our understanding of the landscape of decidable fragments of RML. However, to get a full picture we must also consider when the problem becomes undecidable. In this section we consider which type sequents are expressive enough for us to prove undecidability. We also return to RML with recursion. Having previously shown that recursive functions of type $\beta \to \beta$ can be included in $\mathsf{RML}_{\text{O-Str}}$ while preserving decidability, we present a proof (due to Murawski) that allowing recursive functions of type $(\beta \to \beta) \to \beta$ is already too expressive and that observational equivalence is undecidable in their presence.

4.3.1 On the Right-Hand Side

We first consider which types are sufficient on the right-hand side of the turnstile (in recursion-free RML) to show undecidability. In [72] it is shown that observational equivalence is undecidable for fifth-order terms. The proof takes the strategy that was used to show undecidability for fourth-order IA (by encoding a Turing powerful automaton) and finds an equivalent call-by-value strategy. This relies on a *qqqqq*-branch in the prearena. In fact, the first two questions are played only once so it is relatively straightforward to adapt the proof to show that observational equivalence is undecidable at third-order types (e.g. $((\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \to \mathsf{unit}$) as these contain *qaqqq*-branches. A further result in [73] showed that the problem is undecidable at the type $(\mathsf{unit} \to \mathsf{unit}) \to (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$. Both of these results easily generalise to show that observational equivalence is undecidable at *any* third-order type and *any* second-order type which takes at least two first-order arguments.

We will now modify the second of these proofs to extend these results further and show undecidability at $(\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit} \to \mathsf{unit}$. Our proof will easily adapt to any second-order type (of arity greater than one) which contains a first-order argument which is not the final argument.

4.3.1.1 Q-Stores

Following previous game semantics based undecidability results, we will reduce the halting problem for a class of finite state machines equipped with a queue to observational equivalence of RML-terms. The universality of such machines goes back to

Post's work on simple rewriting systems [93, 70]. In particular, we will utilise automata equipped with a Q-store [72]. Q-stores are a generalisation of a queue which do not always follow queue behaviour. However, we will be able to detect whether the queue discipline has been followed correctly or not.

Definition 4.4. A *Q-store* stores characters from a finite alphabet Σ. Its content is defined by a natural number n and a function $f : \{0, \ldots, n\} \to \Sigma \times \{+, -\} \times \{+, -\}$. The three fields of $f(i)$ will be referred to as $f(i).SYMBOL$, $f(i).ACCESSED$ and $f(i).MARKED$ respectively. The first holds the character stored in this element of the Q-store and the other two are used for bookkeeping.

The empty Q-store is defined by $n = 0$ and $f(0) = (\dagger, +, -)$ where $\dagger$ is a dummy symbol set as accessed but unmarked.

There are two operations which can be performed on a Q-store.

- ADD x adds $x \in \Sigma$ to the store. The new Q-store $f' : \{0, \ldots, n+1\} \to \Sigma \times \{+, -\} \times \{+, -\}$ is defined by $f \subseteq f'$, $f'(n+1) = (x, -, -)$.

- FETCH is the only access method. It can return any previously unaccessed element in the store $f(i).SYMBOL$ (i.e. $f(i).ACCESSED = -$) provided an index j can be found such that $0 \leq j < i \leq n$, $f(j).ACCESSED = +$ and $f(j).MARKED = -$. As well as returning the value stored in the ith element, the operation sets $f(i).ACCESSED$ and $f(j).MARKED$ to $+$.

We see that a FETCH operation can return any unaccessed element i provided there is an earlier element j which has already been accessed but has not yet been marked. The choice of (i, j) is made nondeterministically and different choices can affect the store in different ways. It is possible that the Q-store might behave as a queue. This will occur if during a FETCH the choice of i will always be the first unaccessed element and j to be $i - 1$. If this happens then the Q-store will have a characteristic pattern: no unaccessed element occurs between two accessed elements. The only way to have a Q-store with this pattern is if its behaviour has been that of a queue. In particular, if all elements of a Q-store have been accessed then its behaviour was that of a queue.

We can now consider finite state machines equipped with Q-stores.

Definition 4.5. A *Q-machine* is a tuple $\mathcal{A} = \langle Q, \Sigma, q_0, F, \delta^{ADD}, \delta^{FETCH} \rangle$, where:

- $Q = Q^A + Q^F + F$ is the finite set of states with $q_0 \in Q$ the initial state.

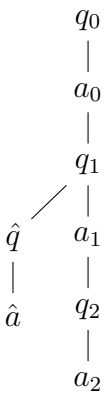


Figure 4.6: Prearena for $\vdash (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit} \to \mathsf{unit}$

- $\delta^{ADD} : Q^A \to Q \times \Sigma$ defines transitions out of states in Q^A. If the machine is in state q_1 and $\delta^{ADD}(q_1) = (q_2, a)$ then the machine transitions into state q_2 and performs ADD a on the machine's Q-store.
- $\delta^{FETCH} : Q^F \times \Sigma \to Q$ defines the machine's action when in a state from Q^F. When in state $q_1 \in Q^F$ the Q-machine will attempt to perform a FETCH. If this is successful and returns symbol a then the machine transitions into state $\delta^{FETCH}(q_1, a)$.

We say that a Q-machine *halts* if there exists a run (starting in the initial state) which ends in a final state (a state in F) with a Q-store in which all elements have been accessed.

Since Q-machines only halt when every element in the Q-store has been accessed (so when the Q-store has acted as a queue) as far as halting is concerned they are the same as finite state automata equipped with a queue. Hence, from Post's work we can infer that they have an undecidable halting problem.

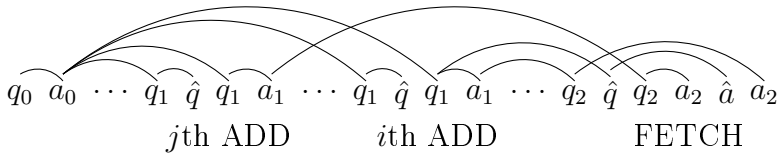
4.3.1.2 Representing Q-machines

We now consider how to represent the run of an arbitrary Q-machine at the type sequent $\vdash (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit} \to \mathsf{unit}$. The relevant prearena is shown in Figure 4.6. For technical convenience we will assume that the initial state of the Q-store results from a dummy ADD action executed once at the very start of the run.

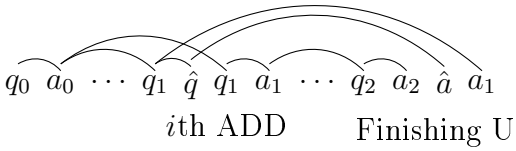
Our representation of the Q-machine will begin with $\widehat{q_0\ a_0}$.

Each ADD operation (including the dummy operation initializing the store) will then be interpreted by the segment $\widehat{q_1\ \hat{q}}\ \widehat{q_1\ a_1}$.

Each FETCH will be represented by segments $q_2\ \hat{q}\ q_2\ a_2\ \hat{a}\ a_2$ where the first q_2 is justified by the a_1 from the ith ADD, $\hat{q}$ is justified by the q_1 immediately before that a_1 and the second q_2 is justified by the a_1 in the jth ADD. Here we are using the visibility condition to force the choice of j to be a strictly earlier ADD-block than the choice of i.



Once the Q-machine has reached a final state at the end of the computation, we must check that the Q-store has the correct shape. This is performed in a finishing up state where we visit each ADD-block from last to first and check each of them has been accessed.



In order to construct a term which follows this strategy we first consider some terms which perform the various responses. Our final term will keep track of which state the simulation is in and imitate one of these terms accordingly.

- $\lambda f. \ldots$ will respond to the initial q_0 with a_0.
- $\lambda f. f(); \lambda x. \Omega$ responds to q_1 with $\hat{q}$. Once this is (eventually) answered with $\hat{a}$ it responds with a_1. This a_1 can never be used to justify anything or else P will not respond.
- $\lambda f. \lambda x. f()$ responds to q_1 with a_1. If this a_1 is used to justify a q_2 then it responds with $\hat{q}$. If this is answered with $\hat{a}$ then it responds with a_2.
- $\lambda f. \lambda x. ()$ responds to q_1 with a_1 and to q_2 with a_2.

In order to keep track of which stage of the computation we are in, we will use a number of global variables.

- *State* — keeping track of which state the simulated Q-machine is in.

- *First* — a flag letting us know if the first dummy ADD-operation has occurred.
- *AddState* — keeping track of how far through an ADD-operation we are.
- *FetchState* — keeping track of how far through a FETCH-operation we are.
- *FinishingState* — keeping track of how far through a finishing up operation we are.

Additionally, we will create several local variables for each ADD.

- *Symbol*, *Accessed* and *Marked* — representing the appropriate fields in the Q-store.
- *Finalised* — a flag keeping track of whether this ADD-operation has been visited during the finishing up stage. This is needed to ensure that each ADD is visited exactly once during this phase.

The term is shown in Figure 4.7. We use the syntax $[B_1, \ldots, B_n]$ as an abbreviation for $\mathbf{if} \bigwedge B_i \mathbf{then} () \mathbf{else} \Omega$. The local variables are associated with the $q_1 \cdot a_1$ part of each ADD-block. This ensures they can be accessed during a FETCH or the finishing up stage when moves are hereditarily justified by them. Note that we cannot enforce that during the finishing up stage, the q_2 is justified by the last unfinalised a_1. However, we do ensure that each a_1 justifies at most one q_2 during this phase. Since we can rely on the second part of the finishing up state ($\hat{a} \cdot a_1$) to hide (by visibility) the a_1 from the last (by bracketing) unfinalised ADD-block, we know that the only way to reach a complete play is if O does indeed finalise the ADD-blocks in order from last to first.

To establish undecidability we note that the represented Q-machine will halt if and only if the term is not observationally equivalent to $\lambda f.\Omega$. Hence, observational equivalence is undecidable if the type contains a first-order (or higher) argument which is not the final argument (i.e. any type of the form $\theta_n \to \ldots \to \theta_4 \to (\theta_3 \to \theta_2) \to \theta_1 \to \theta_0$ for any RML types θ_i and $n \geq 3$).

4.3.2 On the Left-Hand Side

We now consider which types are sufficient to show undecidability on the left-hand side of the turnstile. Note that $\vdash M \cong N : \theta$ if and only if $f : \theta \to \mathsf{unit} \vdash fM \cong fN : \mathsf{unit}$. Thus, for any type sequent $\vdash \theta$ at which observational equivalence is undecidable, we know that the type sequent $\theta \to \mathsf{unit} \vdash \mathsf{unit}$ is also undecidable. So on the left-hand

```
1  let
2    State = ref q_0
3    First = ref 1
4    AddState = ref 0
5    FetchState = ref 0
6    FinishingState = ref 0
7  in
8    λ f .
9      [!State ∈ Q^A];
10     if !AddState = 0 then
11       AddState := 1;
12       f();
13       [!State ∈ F, !FinishingState = 1];
14       FinishingState := 0;
15       λ x . Ω
16     else if !AddState = 1 then
17       let
18         Symbol = ref ‡
19         Accessed = ref (if !First then + else −)
20         Marked = ref −
21         Finalised = ref −
22       in
23         AddState := 0;
24         if !First then
25           First := 0; Symbol := †;
26         else
27           (Symbol, State) := δ^ADD(!State);
28         λ x .
29           if !State ∈ Q^F then
30             if !FetchState = 0 then
31               [!Accessed = −];
32               Accessed := +; FetchState := 1;
33               f();
34               [!FetchState = 2];
35               FetchState := 0; State := δ^FETCH(!State, !Symbol);
36             else if !FetchState = 1 then
37               [!Accessed = +, !Marked = −];
38               FetchState := 2; Marked := +;
39             else Ω
40           else if !State ∈ F then
41             [!FinishingState = 0, !Accessed = +, !Finalised = −];
42             FinishingState := 1; Finalised := +;
43           else Ω
44     else Ω
```

Figure 4.7: The term encoding a Q-machine

side of the turnstile we know the problem is undecidable if we have any fourth-order types or any (third-order) type which has a second-order argument whose first-order argument is not the last.

4.3.3 Recursion

In IA, observational equivalence becomes undecidable if we add recursive first-order functions. The proof appears in [86] and is based on a result from [55]. Murawski has shown how to adapt this proof to RML with recursive functions of type $(\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$. We briefly review his proof.

We again rely on finite state systems equipped with a queue. However, rather than rely on Q-machines, this time we utilise a programming system called *Queue*.

Definition 4.6. A *Queue program* has a single memory cell z that can store a symbol from Σ and a queue (which can contain symbols from Σ). A program consists of a finite sequence of instructions of the form $1 : I_1, 2 : I_2, \ldots, m : I_m$, where each I_i is one of the following:

- `enqueue` a: add the symbol $a \in \Sigma$ to the end of the queue and go to the next instruction.
- `dequeue`: if the queue is empty then halt, otherwise remove the element at the front of the queue and store it in z then go to the next instruction.
- `if` $z = a$ `goto` L where $a \in \Sigma$ and $L \geq 0$ is a label. If the value stored in z is a then go to the Lth instruction, otherwise go to the next instruction.
- `halt`.

The halting problem for Queue programs is undecidable [55].

We will simulate Queue programs using a recursive function of type $(\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$. We will model the queue using the call-stack. Every `enqueue` will cause a recursive call which will allocate a variable *cur* containing the value to be enqueued. When an item is removed from the queue we will set *cur* to 0 which we assume is a special value not in Σ. This means that we know that the head of the queue corresponds to the oldest recursive call whose *cur* does not contain 0.

In addition to the local variable *cur* we will also need global variables *halt* (a flag letting us know we should stop the computation and collapse the call-stack), *pc* (which instruction we are currently on), z (the Queue program's memory cell) and

I_i	J_i
`enqueue` n	$pc := i+1$; $G := n$; $F(\lambda x.\mathbf{if}\ !H = 0\ \mathbf{then}\ L\ \mathbf{else}\ R)$ where $L \equiv G := !cur$ $R \equiv \mathbf{if}\ (H := 0; arg(); !G = 0)\ \mathbf{then}\ z := cur;\ cur := 0$ $\quad\quad \mathbf{else}\ H := 1; arg()$
`dequeue`	$\mathbf{if}\ !cur = 0\ \mathbf{then}\ halt := 1\ \mathbf{else}$ $\quad\quad \mathbf{if}\ H := 0; arg(); !G = 0\ \mathbf{then}\ z := !cur;\ cur := 0\ \mathbf{else}$ $\quad\quad\quad H := 1; arg();$ $\quad\quad pc := i+1$
`halt`	$halt := 1$
`if` $z = n$ `goto` L	$\mathbf{if}\ !z = n\ \mathbf{then}\ pc := L\ \mathbf{else}\ pc := i+1$

Table 4.1: Simulations for each Queue program instruction

two variables G and H. When we make our recursive call, the new value to be added to the queue will be (temporarily) stored in G. Further, the argument to the call (a function of type $\mathsf{unit} \to \mathsf{unit}$) will be such that if it is run when $H = 0$ then the value of cur from the previous call will be written to G. If, on the other hand, the argument is run when $H = 1$ it will cause the value at the front of the queue to be written to G and the appropriate cur to be set to 0 (i.e. that element is removed from the queue).

Our term encoding a queue program is then

$$\begin{aligned}&\mathbf{let}\ halt, pc, z, G, H = \mathbf{ref}\ 0, \mathbf{ref}\ 1, \mathbf{ref}\ 0, \mathbf{ref}\ 0, \mathbf{ref}\ 0\ \mathbf{in}\\ &\qquad (\mu F^{(\mathsf{unit}\to\mathsf{unit})\to\mathsf{unit}}.\lambda arg^{\mathsf{unit}\to\mathsf{unit}}.body)(\lambda c^{\mathsf{unit}}.\Omega)\end{aligned}$$

where *body* has the form

$$\mathbf{let}\ cur = \mathbf{ref}\ (!G)\ \mathbf{in}\ \mathbf{while}\ !halt = 0\ \mathbf{do}\ \mathbf{case}(!pc)[1 \mapsto J_1, \ldots, m \mapsto J_m].$$

Each J_i depends on I_i according to Table 4.1. This term is equivalent to $\vdash ()$ if and only if the simulated Queue program halts. Hence, observational equivalence of $\mathsf{RML}_{\text{O-Str}}$ with recursive functions of type $(\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$ is undecidable.

4.4 Summary

In the previous chapter we considered the O-strict fragment of RML which consisted of terms of short types (order at most two, arity at most one) which could contain

free identifiers with short arguments. We showed that observational equivalence was decidable and ExpTime-complete for this fragment. In this chapter we considered extensions to this fragment. First we showed that adding recursive functions of order one and arity one still preserves decidability but that the power of DPDA is needed; VPA are no longer expressive enough. In Section 4.3.3 we saw that allowing recursive second-order functions is already sufficient for the problem to become undecidable.

We also considered non-O-strict type sequents. Encoding the location of O-pointers is a lot harder than encoding P-pointers as they are controlled by the environment rather than the term. As observational equivalence is defined by a quantification over all contexts, this means that the strategy for a term may have to consider all legal locations of O-pointers rather than just a single location as is the case for P-pointers. This means that representing non-O-strict strategies as languages seems to require an infinite alphabet. One class of automata which recognise languages over infinite alphabets are CMA and we showed that terms of order one and arity two (whose free variables can take arguments of order one and arity one) can be represented as deterministic CMA. Hence, they have a decidable observational equivalence problem. However, many non-O-strict types are undecidable. At third-order types and any type which takes a first-order argument which is not the last argument (and any type sequent with free identifiers which accept arguments of such types) we can show undecidability. Our proofs take advantage of the location of O-pointers to encode the Q-store actions of a Q-machine.

Unfortunately, though close, we still do not have a complete picture of the decidable fragments of RML. There are still cases for which we do not know whether the problem is decidable or not. We could represent terms of type $\beta \to \beta \to \beta$ as CMA since λ-abstracting to reach this type requires interleaving plays of type $\beta \to \beta$ which can be represented as regular expressions. Ensuring each thread is in its own class allowed us to not only determine the position of justification pointers but also to keep track of the state of each thread. If we move to type $\beta \to \beta \to \beta \to \beta$, it appears that we would need to interleave threads consisting of data words. This is much harder to do in the same way and it seems this might require some form of nested data values. There has been some work on words with nested data [20] but it is not yet clear whether we can obtain a similar decidability result using this method. A similar problem occurs if we allow free identifiers to have more complex types. Representing the type sequent $(\beta \to \beta \to \beta) \to \beta \vdash \beta \to \beta \to \beta$ requires encoding the location of O-pointers on both the left- and right-hand side of the turnstile. This again appears to require data words with nested data.

A different problem occurs when we consider terms of type $\beta \to (\beta \to \beta) \to \beta$. Here the game semantics consists of interleaved threads, each of which can be represented as a VPA. The VPA stack seems necessary as an arbitrary number of questions can be open at once and we would need to match them all with answers. However, an infinite alphabet also appears to be required to encode the location of O-pointers. We currently do not know whether a form of Visibly Pushdown CMA could be used which would be expressive enough to capture the desired languages and still have a decidable equivalence problem. This problem also occurs at type sequents such as $((\beta \to \beta) \to \beta) \to \beta \vdash \beta \to \beta \to \beta$ where we need to encode O-pointers on the right-hand side, but the free identifiers can lead to an arbitrary number of questions being open.

While we currently do not know how to show decidability at these type sequents, we also cannot adapt our existing undecidability results to these type sequents either. In our representation of Q-machines we had sequences of moves representing ADD-operations and FETCH-operations. A crucial part of the encoding was that during a FETCH, O had to select two ADD-blocks and P could use visibility to force to the second one to have occurred earlier than the first one. Unfortunately, at these sequents it does not appear possible to enforce this.

Finally, we have not got a full picture of recursion either. We know the problem is decidable for (O-strict RML with) first-order arity one recursive functions. Further, it is undecidable when second-order functions are allowed. Allowing first-order recursive functions with arity greater than one again seems to require combining pushdown automata (in this case not Visibly Pushdown) with CMA.

A summary of the different fragments and some representative type sequents is shown in Table 4.2. We also show what level of recursion is allowed in each fragment. We write Ω to mean that an undecidability result holds (or a result is unknown) even if no recursion or loops are present and non-termination is introduced only through the constant Ω.

In the next chapter we consider languages with non-local control flow. In particular, we will examine call-by-name IA augmented with **catch**. We will see that the addition of such a construct does not affect the decidability result for IA_3^*.

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)	$((\beta \to \ldots \to \beta) \to \beta) \to \ldots \to \beta \vdash (\beta \to \ldots \to \beta) \to \beta$	$\beta \to \beta$
$\mathsf{RML}_{\mathrm{CMA}}$	$(\beta \to \beta) \to \ldots \to \beta \vdash \beta \to \beta \to \beta$	**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	$(\beta \to \beta) \to \beta$
Unknown		
Visibly Pushdown CMA	$\vdash \beta \to (\beta \to \beta) \to \beta$ $((\beta \to \beta) \to \beta) \to \beta \vdash \beta \to \beta \to \beta$	Ω
$\mathsf{RML}_{\mathrm{CMA}}$ + Recursion	$(\beta \to \beta) \to \ldots \to \beta \vdash \beta \to \beta \to \beta$	$\beta \to \beta \to \beta$
Nested Data	$\vdash \beta \to \beta \to \beta \to \beta$ $(\beta \to \beta \to \beta) \to \beta \vdash \beta \to \beta \to \beta$	Ω

Table 4.2: Summary of decidability results for RML

Chapter 5

Non-Local Control Flow

In the previous two chapters we have considered the language RML and for which fragments its observational equivalence problem is decidable. RML is a language which is both higher-order and stateful. It does not however, contain any constructs which allow non-local jumps of control to be made. By contrast, most programming languages do allow such jumps through various constructs such as exceptions, **goto** or **call/cc**. In this chapter we consider languages with non-local control flow, primarily in the context of call-by-name IA.

The first to study control operators in the context of game semantics was Laird [60, 61]. He showed that adding **call/cc** to PCF corresponded to dropping the game-semantic bracketing condition of [52]. He also showed that in the absence of the bracketing rule, the visibility rule still implies a "Weak Bracketing Condition" (as long as no answer ever justifies a question):

> "Only open questions may be answered. A question is open if it is unanswered, and no questions asked before it have been answered since it was asked."

If as well as dropping bracketing we additionally remove the innocence condition (moving along a different axis of the semantic square [8]) then we get a model of Idealized Algol with control. This language is $\mathrm{IA}_{\mathbf{catch+mkvar}}$, which is IA (with a bad variable constructor) extended with a control operator:

$$\frac{\Gamma, x : \mathsf{com} \vdash M : \mathsf{com}}{\Gamma \vdash \mathbf{catch}\, x\, \mathbf{in}\, M : \mathsf{com}}$$

This term will behave as M, unless M attempts to run x at which point control jumps out of the block and terminates. It is a simple yet expressive control operator

which can be used to encode more complex constructs such as those of SPCF [26]. The game semantics is defined in the same manner as for IA without **catch** except that plays can violate bracketing and **catch** is interpreted using the strategy $\mathsf{catch} : [\![(\mathsf{com}_2 \to \mathsf{com}_1) \to \mathsf{com}_0]\!]$ which responds to the initial move run_0 with run_1 and to O's second-move (regardless of whether it is a run_2 or a done_1) with done_0.

The model is fully abstract and has the nice property that for any $\mathrm{IA}_{\mathbf{catch+mkvar}}$ terms $\Gamma \vdash M_1, M_2 : T$,

$$\Gamma \vdash M_1 \sqsubseteq_{\approx \mathrm{IA}_{\mathbf{catch+mkvar}}} M_2 \iff [\![\Gamma \vdash M_1]\!] \subseteq [\![\Gamma \vdash M_2]\!] \text{ [75]}.$$

Note that this relation on strategies compares all even-length plays, rather than just complete plays as is the case with IA without **catch**.

Example 5.1. Consider the two terms $x : \mathsf{com} \vdash \Omega$ and $x : \mathsf{com} \vdash x; \Omega$. The strategies denoting these terms in the game-semantic model (of both IA and $\mathrm{IA}_{\mathbf{catch+mkvar}}$) are $\{\, \epsilon \,\}$ and $\{\, \epsilon, \mathsf{run} \cdot \mathsf{run}_x \,\}$ respectively. Since neither of these strategies contain any complete plays, the two terms are equivalent in IA. This should be expected as if either the terms is ever run it will not terminate. Despite the fact that the sets of complete plays are equal, the two strategies are clearly not equal. Hence, in $\mathrm{IA}_{\mathbf{catch+mkvar}}$ the two terms are inequivalent. They are simple to separate using the context $C[-] \equiv \mathbf{catch}\, x\, \mathbf{in} - $. This will diverge if given the first term but terminate when given the second.

We know that the third-order fragment of IA (without **catch**) has a decidable observational equivalence problem. We aim to show that this is still the case when non-local control flow is allowed. That is, we will show that the third-order fragment of $\mathrm{IA}_{\mathbf{catch+mkvar}}$ is decidable.

5.1 Third-Order $\mathrm{IA}_{\mathbf{catch+mkvar}}$

5.1.1 Representation

We aim to prove that the third-order fragment of $\mathrm{IA}_{\mathbf{catch+mkvar}}$ is decidable. As with previous decidability results, we will represent the strategy denotations of terms as languages. To do this, we need to be able to encode the location of justification pointers. Recall that arenas in this fragment will have the form shown in Figure 5.1. An ith-order arena will contain only those moves q_j and a_j for $j \leq i$. We first consider all the moves in arenas in this fragment and whether their justification pointers are uniquely reconstructible.

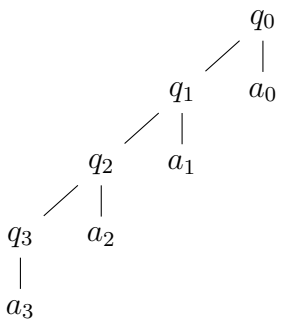


Figure 5.1: Shape of third-order arenas

- The initial move, q_0, does not require a justification pointer.
- First-order questions, q_1, and ground answers, a_0, are justified by the initial move. Since the initial move is unique these pointers are uniquely reconstructible.
- Since first-order questions are justified by the initial move, there can only ever be one in the O-view (as when constructing the O-view we would follow the pointer back to the initial move and no moves precede the initial move). This implies that second-order questions, q_2, and first-order answers, a_1, have uniquely reconstructible justification pointers.
- Sadly, second- and third-order questions are not always unique in the view and so pointers from third-order moves (both questions, q_3, and answers, a_3) and second-order answers, a_2, are not always unique. For example, the standard example showing that third-order questions require explicit justification pointers is the pair of plays $q_0\ q_1\ q_2\ q_1\ q_2\ q_3$ and $q_0\ q_1\ q_2\ q_1\ q_2\ q_3$, which are differentiated only by the location of the final pointer.

We consider below how we can encode the location of the justification pointers where necessary.

5.1.1.1 Third-Order Questions

The problematic third-order questions occur in the well-bracketed case too. A method of encoding them is presented in [81, 76]. An alternative (and perhaps simpler) method was mentioned in Section 3.2.1. That is, each third-order question is tagged with a natural number. A tag of i indicates that this question is hereditarily justified

by the ith oldest open first-order question belonging to a second-order identifier. While this potentially needs an infinite number of indices, for any given term the number needed will be finite. This is because the maximum number required will be the maximum nesting depth of calls to second-order identifiers. This method is also easy to implement. All third-order questions are initially tagged with 1. Then, whenever we see a term of the form fM, where f is a second-order identifier, we increment all the tags coming from our representation of $[\![M]\!]$.

5.1.1.2 Bracketing Violations

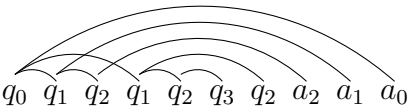
The remaining cases where the locations of pointers are ambiguous are second- and third-order answers. In the well-bracketed case these are never ambiguous because they must answer the most recently asked unanswered question. In the absence of bracketing we need to know how many questions an answer closes in order to count back and find its justifier. The following proposition simplifies this by allowing us to only record the number of *second-order* questions an answer closes.

Proposition 5.1. *To determine the location of the justification pointer for any answer in a strategy for third-order $IA_{\mathbf{catch+mkvar}}$, it suffices to know how many second-order questions the answer closes.*

Proof. For ground and first-order answers we already know that their pointers are uniquely determined. For second-order answers we can immediately count back the appropriate number of open second-order questions to find our justifier. To establish the result for third-order answers first note that the only questions O can play are the initial move (which can only be played once) and second-order questions. Hence, whenever P plays a third-order question, q_3, O must either play an answer, closing (although not necessarily answering) P's question, or a second-order question, q_2. The only way to close q_2 without closing q_3 is for P to answer q_2. However, if this occurs then we are back in a situation where O must either play an answer and close q_3 or play another second-order question. From this it follows that between any two open third-order questions there must be at least one open second-order question. Hence, if we know how many second-order questions a third-order answer closes, then we can uniquely find its third-order question justifier by counting back. □

We can now give our encoding of plays as words. A play is represented by its underlying move sequence, with all third-order questions tagged as explained above. Further, before any answer which closes (but does not answer) n second-order questions, we insert n copies of the new letter break. This converts a play s into a

word $\hat{s}$ in such a way that the original play can be uniquely reconstructed. That is, $s = t \iff \hat{s} = \hat{t}$. This allows us to represent a strategy σ by an automaton which accepts all words $\hat{s}$ where $s \in \sigma$.

Example 5.2. The play  $q_0\ q_1\ q_2\ q_1\ q_2\ q_3\ q_2\ a_2\ a_1\ a_0$ would be represented as the word $q_0 \cdot q_1 \cdot q_2 \cdot q_1 \cdot q_2 \cdot q_3^2 \cdot q_2 \cdot \mathsf{break} \cdot \mathsf{break} \cdot a_2 \cdot a_1 \cdot a_0$.

5.1.2 Construction

Having decided on our encoding of strategies as languages, we can now start to construct automata to accept them. Our construction will closely follow the construction used to show decidability of third-order IA without non-local control flow [81]. As in that case, we will represent strategies using VPA. However, we will partition our VPA alphabets differently. As might be expected from the way we are representing strategies, second-order questions will be push-letters whereas second-order answers and breaks will be pop-letters. All other moves will be noop-letters. In [81] the VPA instead push and pop on third-order questions and answers respectively.

Since we are in a call-by-name setting, it is sufficient to consider only β-normal forms. In a similar manner to [81], this leaves us only needing to be able to handle:

- The special strategies used to interpret the IA constructions (while, seq, etc.).
- Identity strategies $\mathsf{id}_{[\![\theta]\!]}$ ($\mathbf{ord}(\theta) \leq 2$).
- Composition of strategies $\sigma : [\![\theta_1]\!] \Rightarrow [\![\theta_2]\!]$ and $\tau : [\![\theta_2]\!] \Rightarrow [\![\theta_3]\!]$ when either:
  - $\mathbf{ord}(\theta_2) = 0$,
  - $\mathbf{ord}(\theta_2) = 1$ and $\tau \in \{\, \mathsf{cell}, \mathsf{mkvar}, \mathsf{catch} \,\}$,
  - or $\mathbf{ord}(\theta_2) = 1$ and $\sigma \fatsemi \tau = [\![\Gamma, f \vdash f M_1 \dots M_n]\!]$ for some second-order identifier f and terms $M_1, \dots, M_n$.

We also need to handle pairing and currying, but these are trivial, consisting of union and relabelling respectively. Excluding catch, the strategies for the IA constants and first-order identities will be the same as in the well-bracketed case. This is because P will not violate bracketing and O cannot violate bracketing as the arenas they play over are at most second-order. We know from [42] how to represent the complete plays of these strategies as finite automata. To adapt these automata to our

framework we just need to make minor changes; we must ensure they accept all even length prefixes of complete plays and push/pop a dummy symbol on any second-order questions/answers. Similarly, composition when $\mathbf{ord}(\theta_2) = 0$ or $\tau \in \{\mathsf{cell}, \mathsf{mkvar}\}$ cannot lead to new bracketing violations and so we can reuse the constructions of [81] making only minor adaptations so that the automata fit our framework.

This leaves us just needing to handle catch, second-order identifiers and their applications. It should not be surprising that these are the only cases which significantly differ from the well bracketed construction, as these are the cases where P and O respectively can violate bracketing.

- To construct $\mathcal{A}_{\Gamma \vdash \mathbf{catch}\, x\, \mathbf{in}\, M}$ we start with a copy of $\mathcal{A}_{\Gamma, x:\mathsf{com} \vdash M}$. Whenever we see a transition run_x, we instead wish to perform done. However, we must first empty the stack by performing breaks. It is simple to keep track of when the stack is empty by doubling the number of states and stack symbols and using a marked copy to indicate that the stack is empty and an unmarked copy to indicate it is not.

  That is, given a VPA $\mathcal{A}_{\Gamma, x \vdash M}$ for $[\![\Gamma, x : \mathsf{com} \vdash M : \mathsf{com}]\!]$ we take two copies of all the states of $\mathcal{A}_{\Gamma, x \vdash M}$, marking state s in the second copy with a prime, s'. The initial state will be the initial state in the marked copy. All states which were accepting in $\mathcal{A}_{\Gamma, x \vdash M}$ will still be accepting. Similarly, we take two copies of all the stack symbols in $\mathcal{A}_{\Gamma, x \vdash M}$. We also take three fresh states (1), (2) and (3) with (3) final. Then as our transitions we take:

  - If $s \xrightarrow{m} t$ is a noop-transition in $\mathcal{A}_{\Gamma, x \vdash M}$ not on an x-move then we have $s \xrightarrow{m} t$ and $s' \xrightarrow{m} t'$.
  - If $s \xrightarrow{m/\gamma} t$ in $\mathcal{A}_{\Gamma, x \vdash M}$ then we have $s \xrightarrow{m/\gamma} t$ and $s' \xrightarrow{m/\gamma'} t$.
  - If $s \xrightarrow{m,\gamma} t$ in $\mathcal{A}_{\Gamma, x \vdash M}$ then we have $s \xrightarrow{m,\gamma} t$ and $s \xrightarrow{m,\gamma'} t'$.
  - If $s \xrightarrow{\mathsf{run}_x} t$ in $\mathcal{A}_{\Gamma, x \vdash M}$ then $s \xrightarrow{\epsilon} (1)$ and $s' \xrightarrow{\epsilon} (2)$.
  - $(1) \xrightarrow{\mathsf{break},\gamma} (1)$ and $(1) \xrightarrow{\mathsf{break},\gamma'} (2)$ for all stack symbols γ of $\mathcal{A}_{\Gamma, x \vdash M}$.
  - $(2) \xrightarrow{\mathsf{done}} (3)$.

- The remaining cases are second-order free identifiers and applications of them. As in the well-bracketed cases described in [81, 76], these boil down to the dagger construction (described in Section 2.2.3.1 which promotes a strategy $\sigma : !A \multimap B$ to $\sigma^\dagger : !A \multimap !B$) plus some simple renamings. For example, $\mathsf{id}_{A \Rightarrow B} = \mathsf{id}^\dagger_A \multimap \mathsf{id}_B$

and in the well-bracketed case complete plays of $\Gamma, f : (\mathsf{com} \to \mathsf{com}) \to \mathsf{com} \vdash fM : \mathsf{com}$ are of the form

$$r \cdot r_f \left(\{ \epsilon \} \cup \mathbf{comp}(\llbracket \Gamma, f \vdash M \rrbracket^\dagger) \, [r_{1f}, r_{2f}, d_{1f}, d_{2f}/r, r_1, d, d_1] \right) d_f \cdot d.$$

Since we also need incomplete, non-well-bracketed plays, we will need our automaton to accept all even-length prefixes of plays having the form

$$\begin{aligned} & r \cdot r_f \left(\llbracket \Gamma, f \vdash M \rrbracket^\dagger \, [r_{1f}, r_{2f}, d_{1f}, d_{2f}/r, r_1, d, d_1] \right) \\ & \cup \, r \cdot r_f \left(\{ \epsilon \} \cup \mathbf{answerable}(\llbracket \Gamma, f \vdash M \rrbracket^\dagger) \, [r_{1f}, r_{2f}, d_{1f}, d_{2f}/r, r_1, d, d_1] \right) d_f \cdot d \end{aligned}$$

where $\mathbf{answerable}(\llbracket \Gamma, f \vdash M \rrbracket^\dagger)$ is the set of plays in $\llbracket \Gamma, f \vdash M \rrbracket^\dagger$ which end in d or r_1 (visibility prevents O from answering r_f unless P has played one of these moves). The only complicated part of constructing this is the dagger construction and we discuss this below.

5.1.2.1 The Dagger Construction

We only need the dagger construction for strategies $\sigma : !\llbracket \Gamma \rrbracket \multimap \llbracket \theta \rrbracket$ where θ is a first-order type. We consider what this construction will look like. We let q_0 and q_1 range over questions of order zero and one from $\llbracket \theta \rrbracket$ and a_0, a_1 over their respective answers.

For a strategy $\sigma : !\llbracket \Gamma \rrbracket \multimap \llbracket \theta \rrbracket$, its promotion $\sigma^\dagger : !\llbracket \Gamma \rrbracket \multimap !\llbracket \theta \rrbracket$ is defined as $\{ s \in P_{!\llbracket \Gamma \rrbracket \multimap !\llbracket \theta \rrbracket} \mid \text{for all initial } m,\ s \upharpoonright m \in \sigma \}$. So $\sigma^\dagger$ appears to be a simple interleaving of plays from σ. However, the restriction that $s \in P_{!\llbracket \Gamma \rrbracket \multimap !\llbracket \theta \rrbracket}$ and in particular that $s \upharpoonright \llbracket \Gamma \rrbracket$ is a valid play on $!\llbracket \Gamma \rrbracket$ and $s \upharpoonright \llbracket \theta \rrbracket$ is a valid play on $!\llbracket \theta \rrbracket$ enforces a switching condition: only O can switch between (or create new) threads of σ and further O can only do this after P plays in $\llbracket \theta \rrbracket$ (which must be either q_1 or a_0). When P plays such a move, the O-view will have the form $(q_0 \cdot (q_1 + a_0))^*$ (i.e. only moves from $\llbracket \theta \rrbracket$ will be visible). Further, an unanswered q_1 from every open σ thread will be visible (since a thread can only be left after P plays q_1 or a_0, every open thread will have an unanswered q_1 and the only way an unanswered q_1 could not be present in the view is if that thread was closed by O playing a_1 to return to an earlier thread). So whenever P plays q_1 or a_0, O has the option of either opening a new σ-thread with q_0 or answering any currently open q_1, possibly violating bracketing.

Now we consider how to construct a VPA $\mathcal{A}_{\sigma^\dagger}$ to recognise $\sigma^\dagger$ from the VPA $\mathcal{A}_\sigma$ which recognises σ. Similarly to [81] and [76], we use a slightly different scheme for the

automata resulting from the dagger construction to those we use for representing denotations of terms. In particular, we partition our VPA alphabets slightly differently. We will continue to push and pop on second-order moves (from $[\![\Gamma]\!]$) and to count the number of such questions closed by bracketing violations using break. However, we will also push, pop and count closures for order zero $[\![\theta]\!]$-moves (which are noop-letters in $\mathcal{A}_\sigma$). The rationale behind this is that after we use the dagger construction, the next step will always involve renaming $[\![\theta]\!]$-moves in a way which increases their order by two (as in the renamings presented above for $[\![\Gamma, f \vdash fM]\!]$). Thus, once the renaming is performed, the VPA will have the desired alphabet partitioning.

We can now describe the construction. We start with $\mathcal{A}_\sigma$. We will have fresh stack symbols s_2 for each state s_2 with an incoming transition $s_1 \xrightarrow{q_1} s_2$. These will be pushed on to the stack when we start a new σ-thread in order to remember which state we were in. All our uses of the dagger construction will want the empty word ϵ to be accepted and so we make the initial state accepting. We also add a new stack symbol *INITIAL* to be pushed when opening a σ-thread when there are no other open σ-threads. This will allow us to know when the stack is empty. We then add a new non-final state r_γ for each stack symbol γ (of the newly expanded stack alphabet). These will only be visited after a q_1 to allow O to pop the stack and violate bracketing. We modify the transition relation as follows:

- Each initial transition $i \xrightarrow{q_0} j$ needs to be changed into a push-transition, pushing the new symbol *INITIAL* to indicate this is the first symbol on the stack, $i \xrightarrow{q_0/INITIAL} j$.

- For each transition $s_1 \xrightarrow{q_1} s_2$ and initial transition $i \xrightarrow{q_0} j$ we add transitions $s_2 \xrightarrow{q_0/s_2} j$ and for each stack symbol γ the transition $s_2 \xrightarrow{\mathsf{break},\gamma} r_\gamma$. That is, after a q_1 move, as well as being allowed to continue in the same thread as before, we also allow O to start a new thread, remembering the state of the old one on the stack, or to violate bracketing in order to return to an old thread.

- For each pair of transitions $s_1 \xrightarrow{q_1} s_2 \xrightarrow{a_1} s_3$ we add transition $r_{s_2} \xrightarrow{a_1} s_3$. So if O is violating bracketing and the last question which we pop off the stack is a q_0 then we can return to the thread which was running before that q_0 was played.

- For each pair of stack symbols γ, γ' we add $r_\gamma \xrightarrow{\mathsf{break},\gamma'} r_{\gamma'}$. So when in r_γ we can continue to pop the stack until we find a previous thread we wish to return to. Note that since each r_γ is non-final we do not have to worry about popping too far — if we get stuck in r_γ the word will not be accepted.

- Finally, we will change each transition $t_1 \xrightarrow{a_0} t_2$ into a pop-transition $t_1 \xrightarrow{a_0, s_2} s_2$ and $t_1 \xrightarrow{a_0, INITIAL} i$. That is, when closing a thread with a_0, we return to whatever state we were in before it was opened.

Example 5.3. To illustrate this construction, we consider the second-order identifier $f : (\mathsf{com} \to \mathsf{com}) \to \mathsf{com} \vdash f$. For clarity we will use r, d and b as abbreviations for the moves run, done and break respectively. As we have previously indicated, we will use the dagger construction and take advantage of the equivalence $\mathsf{id}_{A \Rightarrow B} = \mathsf{id}_A^\dagger \multimap \mathsf{id}_B$. So, our starting point is the first-order identifier $g : \mathsf{com} \to \mathsf{com} \vdash g$. The VPA for this term is shown in Figure 5.2. Using our dagger construction on this VPA results in

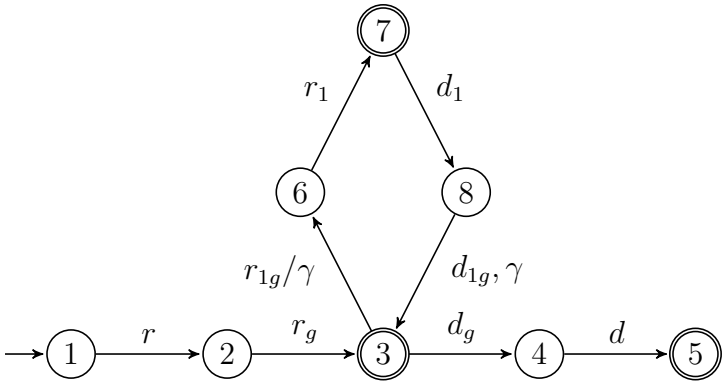


Figure 5.2: VPA for $g : \mathsf{com} \to \mathsf{com} \vdash g$

the VPA shown in Figure 5.3. Finally, we can utilise $\mathsf{id}_{A \Rightarrow B} = \mathsf{id}_A^\dagger \multimap \mathsf{id}_B$ to transform this automaton into a VPA for $f : (\mathsf{com} \to \mathsf{com}) \to \mathsf{com} \vdash f$. This requires a simple relabelling of the moves, prefixing the moves $r \cdot r_f$ to all runs and allowing $d_f \cdot d$ to occur (possibly violating bracketing) after any r_1 or d. We know that when d_f is played the stack should be empty (as it closes any open second-order questions) and that the only states reachable after a r_1 or d (and possibly some number of b's) are states 1, 7 and the r_{sym}'s. Since we want the stack to be empty, we allow d_f to occur from states 1 and r_I. The final automaton is shown in Figure 5.4.

This completes the construction and allows us to represent the strategy denotation of any third-order $\mathrm{IA}_{\mathbf{catch+mkvar}}$-term as a VPA.

Theorem 5.1. *Observational equivalence of the third-order fragment of* $IA_{\mathbf{catch+mkvar}}$ *is decidable.*

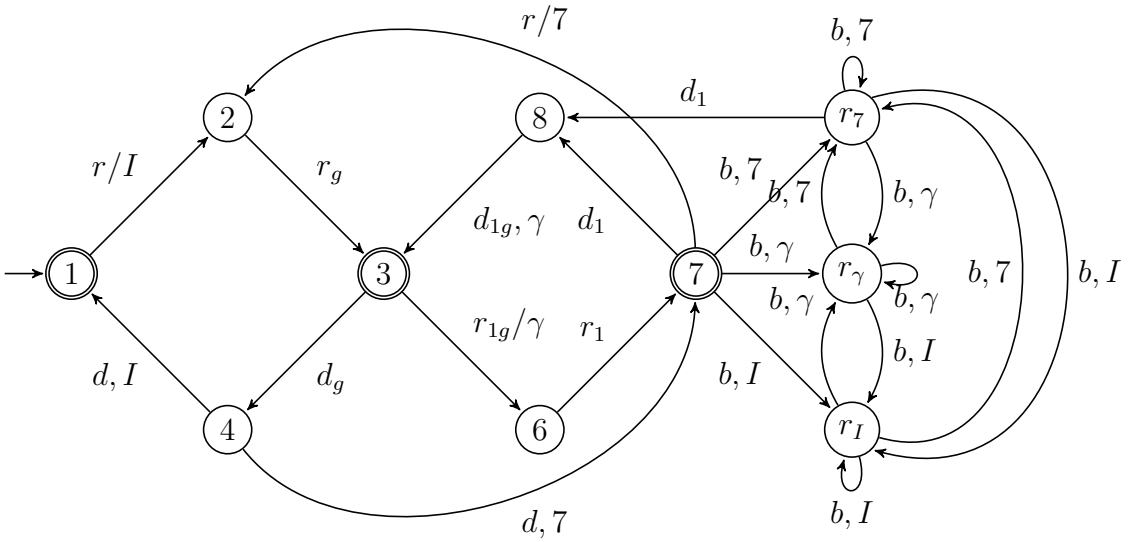


Figure 5.3: VPA for $[\![g : \mathsf{com} \to \mathsf{com} \vdash g]\!]^\dagger$

5.2 Removing mkvar

The results as presented so far apply to the language $\mathrm{IA}_{\mathbf{catch+mkvar}}$ which includes the bad-variable constructor **mkvar**. In the well-bracketed case we know that the inclusion of **mkvar** does not affect the equivalence relation — two **mkvar**-free terms are equivalent in IA if and only if they are equivalent in $\mathrm{IA}_{\mathbf{mkvar}}$ [67]. Sadly this is not the case when we include **catch**. For example, the two terms $x := 0; \Omega$ and $x := 1; \Omega$ are equivalent in $\mathrm{IA}_{\mathbf{catch}}$ but not in $\mathrm{IA}_{\mathbf{catch+mkvar}}$ because the context can instantiate x with a bad-variable and use **catch** to break out of the assignment before control-flow reaches the divergent Ω.

In [75] Murawski defines an alternative ordering on strategies (based on McCusker's relation in [67]) to deal with the presence of **mkvar**.

Definition 5.1. The relation on plays $\diamond_P$ can be defined as $t \diamond_P t'$ if and only if $t = s_1 \cdot q \cdot s_2$ and $t' = s_1 \cdot q' \cdot s_2$ where q, q' are P-questions from the same var component (so they must be either read or $\mathsf{write}(i)$ for some i) which have not been answered in t, t'.

The relation on plays $\triangleleft_P$ is defined as $t \triangleleft_P t'$ if and only if $t = s_1 \; \overbrace{\mathsf{read} \; s_2 \; i} \; s_3$ and $t' = s_1 \; \overbrace{\mathsf{write}(i) \; s_2 \; \mathsf{ok}} \; s_3$, where read and $\mathsf{write}(i)$ are P-moves from the same var-component.

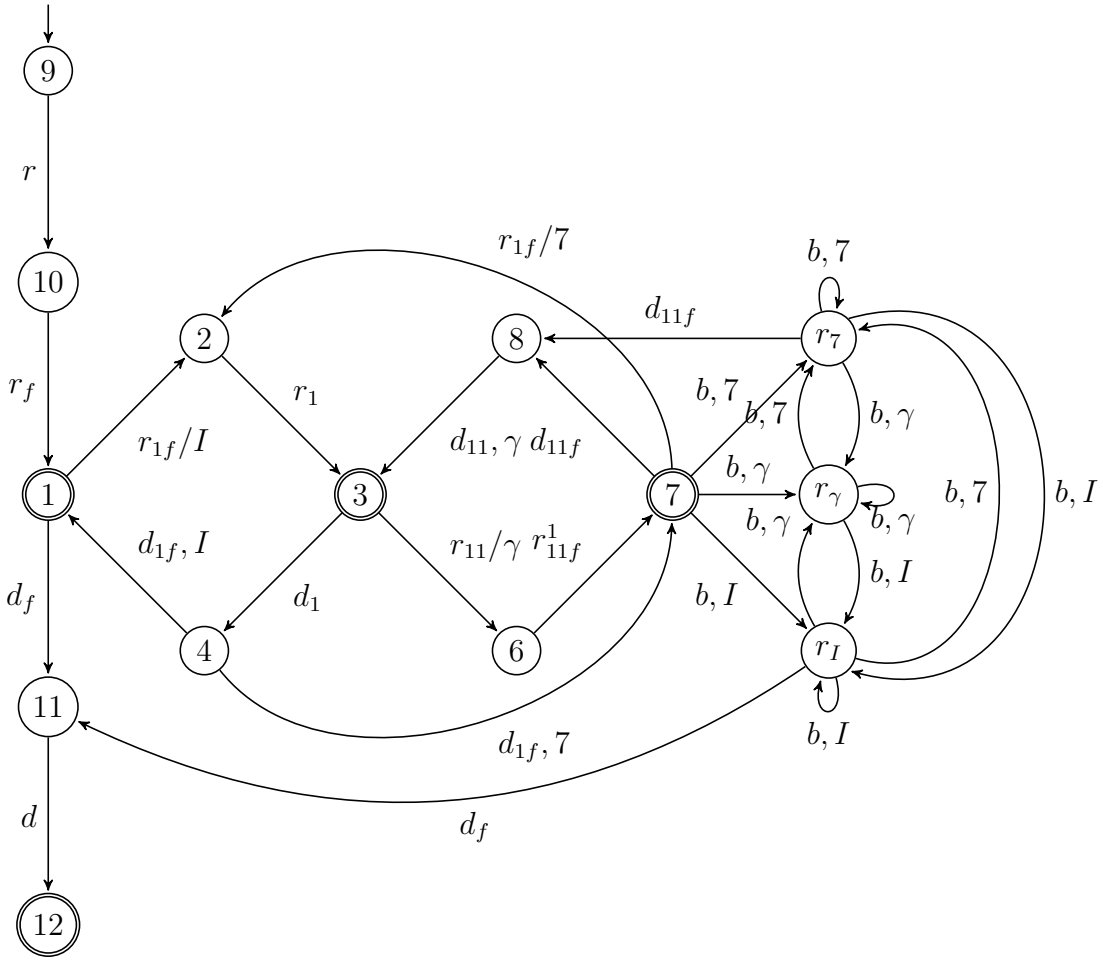


Figure 5.4: VPA for $[\![f : (\mathsf{com} \to \mathsf{com}) \to \mathsf{com} \vdash f]\!]$

Finally we can define the ordering on strategies $\sigma \sqsubseteq \tau$ if and only if for all $s \in \sigma$ there exists $t \in \tau$ such that $s(\diamond_P \cup \triangleleft_P)^* t$.

Under this ordering the game-semantic model for $\mathrm{IA}_{\mathbf{catch}}$ is fully abstract [75]. That is,

$$M_1 \sqsubseteq_{\approx \mathrm{IA}_{\mathbf{catch}}} M_2 \iff [\![M_1]\!] \sqsubseteq [\![M_2]\!].$$

Example 5.4. Returning to the two terms $x := 0; \Omega$ and $x := 1; \Omega$, we have that $[\![x \vdash x := 0; \Omega]\!] = \{\, \epsilon, \mathsf{run} \cdot \mathsf{write}(0) \,\}$ and $[\![x \vdash x := 1; \Omega]\!] = \{\, \epsilon, \mathsf{run} \cdot \mathsf{write}(1) \,\}$. These sets are clearly not equal and so the two terms are not equal in $\mathrm{IA}_{\mathbf{catch+mkvar}}$. However, $\mathsf{run} \cdot \mathsf{write}(0) \diamond_P \mathsf{run} \cdot \mathsf{write}(1)$ (and vice versa) so the two terms are equivalent in $\mathrm{IA}_{\mathbf{catch}}$.

We now consider how to decide this new relation. We will use a slightly different representation of plays. As well as pushing, popping and counting premature closures of second-order moves, we do the same for moves from negative occurrences of var. This means that there may be more breaks than under our previous representation but the location of justification pointers is still uniquely encoded. We also insist that the symbols pushed on var-moves must identify which move they were pushed on (e.g. the symbol pushed could be the same as the input letter).

It is simple to modify our previous constructions to fit this scheme. Changing var-moves to push or pop as appropriate is straightforward. We then just need to deal with break-transitions. These are only introduced in the construction for **catch** and the dagger construction. In **catch** when we start using breaks to pop the stack, we keep going until the stack is empty. This is unaffected by the additional symbols which could be on the stack. Similarly, in the dagger construction we always keep performing breaks until either the stack is empty or a q_0 is popped off the stack. Again this will be unaffected by the possibility of extra symbols on the stack.

Now, given two terms M_1 and M_2 and VPA $\mathcal{A}_{M_1}$ and $\mathcal{A}_{M_2}$ which recognise their denotations, we aim to produce a new automaton $\mathcal{A}_{M_1 \setminus M_2}$ which will accept (representations of) all plays $s \in [\![M_1]\!]$ such that there is no $t \in [\![M_2]\!]$ such that $s(\diamond_P \cup \triangleleft_P)^* t$. Then we will have $[\![M_1]\!] \sqsubseteq [\![M_2]\!]$ if and only if $\mathcal{A}_{M_1 \setminus M_2}$'s language is empty.

We build our automaton using a product construction. States will either be a state from $\mathcal{A}_{M_1}$ or a pair of states from $\mathcal{A}_{M_1}$ and $\mathcal{A}_{M_2}$. We will reach the unpaired states only when we are considering a play s from $[\![M_1]\!]$ for which there is no play t from $[\![M_2]\!]$ such that $s(\diamond_P \cup \triangleleft_P)^* t$. Hence, the final states will be unpaired final states from $\mathcal{A}_{M_1}$. The initial state will be the pair of the initial states. The stack alphabet will be the set of pairs of stack symbols from $\mathcal{A}_{M_1}$ and $\mathcal{A}_{M_2}$. The transition relation will be as follows.

- For moves which are not from a negative occurrence of var we run the two automata in lock step. That is, if $s \xrightarrow{m} s'$ in $\mathcal{A}_{M_1}$ and $t \xrightarrow{m} t'$ in $\mathcal{A}_{M_2}$ then $(s,t) \xrightarrow{m} (s',t')$ in $\mathcal{A}_{M_1 \setminus M_2}$. If m is a push- or pop-letter then we push or pop the pair of the relevant symbols.

- If $\mathcal{A}_{M_2}$ cannot match $\mathcal{A}_{M_1}$ then we allow $\mathcal{A}_{M_1}$ to continue on its own. That is, for a move m which is not from a negative occurrence of var:

  - If $s \xrightarrow{m} s'$ is a noop-transition in $\mathcal{A}_{M_1}$ but there is no transition $t \xrightarrow{m} t'$ in $\mathcal{A}_{M_2}$ then $(s,t) \xrightarrow{m} s'$.
  - If $s \xrightarrow{m/\gamma} s'$ is a push-transition in $\mathcal{A}_{M_1}$ but there is no transition $t \xrightarrow{m/\gamma'} t'$ (for any γ') in $\mathcal{A}_{M_2}$ then $(s,t) \xrightarrow{m/(\gamma,\gamma'')} s'$ for an arbitrary γ''.
  - If $s \xrightarrow{m,\gamma} s'$ is a pop-transition in $\mathcal{A}_{M_1}$ but there is no transition $t \xrightarrow{m,\gamma'} t'$ in $\mathcal{A}_{M_2}$ then $(s,t) \xrightarrow{m,(\gamma,\gamma')} s'$.

- If $s \xrightarrow{q/\gamma} s'$ in $\mathcal{A}_{M_1}$ and $t \xrightarrow{q'/\gamma'} t'$ in $\mathcal{A}_{M_2}$ where q, q' are (possibly different) P-questions from the same $[\![\mathsf{var}]\!]$-component then $(s,t) \xrightarrow{q/(\gamma,\gamma')} (s',t')$. This allows us to handle the case when the two plays are related by $\diamond_P$.

- If $s \xrightarrow{a,\gamma} s'$ in $\mathcal{A}_{M_1}$ and $t \xrightarrow{a',\gamma'} t'$ in $\mathcal{A}_{M_2}$ where a, a' are O-answers from the same $[\![\mathsf{var}]\!]$-component and γ, γ' were pushed by $[\![\mathsf{var}]\!]$ questions q,q' respectively then $(s,t) \xrightarrow{a,(\gamma,\gamma')} (s',t')$ provided that either $q = q'$ and $a = a'$ or for some i we have that $q = \mathsf{read}$, $q' = \mathsf{write}(i)$, $a = i$ and $a' = \mathsf{ok}$. Note that even though the final transition does not mention a', it is uniquely determined by a, γ and γ'. This allows us to handle the case when the two plays are related by $\triangleleft_P$.

- We also need to add transitions for the case when $\mathcal{A}_{M_2}$ cannot match $\mathcal{A}_{M_1}$ on var-moves. So if $s \xrightarrow{q/\gamma} s'$ in $\mathcal{A}_{M_1}$ where q is a P-question from $[\![\mathsf{var}]\!]$ but there is no $[\![\mathsf{var}]\!]$-question q' such that $t \xrightarrow{q'/\gamma'} t'$ (for any γ') in $\mathcal{A}_{M_2}$ then $(s,t) \xrightarrow{q/(\gamma,\gamma'')} s'$ for an arbitrary γ''.

  We have similar transitions for answers. Let γ and γ' be stack symbols pushed by q and q' from a negative occurrence of var, a be an answer from the same var occurrence, s be a state of $\mathcal{A}_{M_1}$ such that $s \xrightarrow{a,\gamma} s'$ and t a state of $\mathcal{A}_{M_2}$.

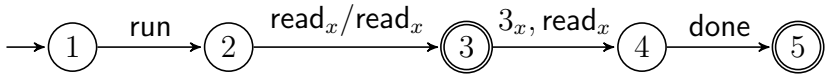
  - If $q = q'$ and there is no transition $t \xrightarrow{a,\gamma'} t'$ in $\mathcal{A}_{M_2}$ then $(s,t) \xrightarrow{a,(\gamma,\gamma')} s'$.
  - If $q = \mathsf{read}$, $q' = \mathsf{write}(i)$, $a = i$ and there is no transition $t \xrightarrow{\mathsf{ok},\gamma'} t'$ in $\mathcal{A}_{M_2}$ then $(s,t) \xrightarrow{a,(\gamma,\gamma')} s'$.

– Finally, if $q \neq q'$ and it is not the case that $q = \mathsf{read}$, $q' = \mathsf{write}(i)$ and $a = i$ then we also have $(s,t) \overset{a,(\gamma,\gamma')}{\rightarrow} s'$.

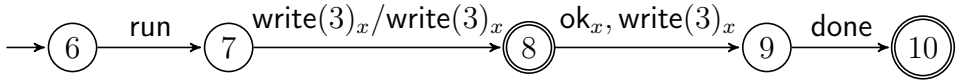
- All the transitions of $\mathcal{A}_{M_1}$ except those on moves from negative occurrences of var will be included unchanged for unpaired states. For those from negative occurrences of var we simply modify them to push and pop pairs of symbols. Pushes should push an arbitrary symbol in the second component, while pops should pop any symbol in the second component.

Example 5.5. Let $M_1 = x : \mathsf{var} \vdash \mathbf{if}\, !x = 3\, \mathbf{then}\, \mathbf{skip}\, \mathbf{else}\, \Omega$ and $M_2 = x : \mathsf{var} \vdash x := 3$. The VPA for these terms are:

- $\mathcal{A}_{M_1}$:

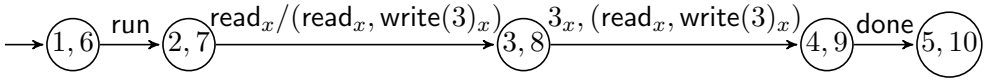


- $\mathcal{A}_{M_2}$:



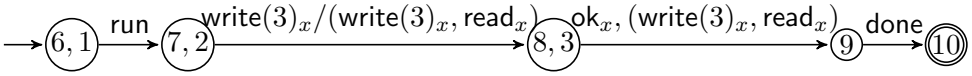
Now we can apply our construction to these two automata.

- $\mathcal{A}_{M_1 \setminus M_2}$:



This automaton has no reachable final states. Hence, its language is empty and so for all $s \in [\![M_1]\!]$ there exists a $t \in [\![M_2]\!]$ such that $s(\diamond_P \cup \triangleleft_P)^* t$, $[\![M_1]\!] \sqsubseteq [\![M_2]\!]$ and $M_1 \sqsubseteq_{\approx \mathrm{IA}_{\mathbf{catch}}} M_2$. This is because M_1 will terminate only if it is placed in a context where x is bound to a (term which evaluates to) a variable currently storing the value 3. If M_2 is placed in the same context then the assignment will have no effect but will also terminate.

- We can also perform the construction the other way around to get $\mathcal{A}_{M_2 \setminus M_1}$:



This time, we can reach an accepting state. The play $s = \mathsf{run} \cdot \mathsf{write}(3)_x \cdot \mathsf{ok}_x \cdot \mathsf{done}$ is in $[\![M_2]\!]$ but there is no play t in $[\![M_1]\!]$ such that $s(\diamond_P \cup \triangleleft_P)^* t$. Hence, we do not have $M_2 \sqsubseteq_{\approx \mathrm{IA}_{\mathbf{catch}}} M_1$ (or $M_1 \cong_{\mathrm{IA}_{\mathbf{catch}}} M_2$). This is easy to see if we consider the context $C[-] = \mathbf{new}\, x\, \mathbf{in} -$ which binds x to a new variable with value 0 and then runs whichever term it is given. When passed M_2 it will terminate, but given M_1 it diverges.

This construction allows us to extend the decidability result of the previous section to IA with **catch** but without a bad-variable constructor.

Theorem 5.2. *Observational approximation and equivalence of the third-order fragment of* $IA_{\mathbf{catch}}$ *is decidable.*

5.3 Summary

In this chapter we have considered decidability results for IA with non-local control flow. Game semantically, allowing non-local control flow corresponds to dropping the bracketing condition. We started out considering third-order IA with both **catch** and **mkvar**. Since the game-semantic model still satisfies a weak bracketing condition, it was possible to model bracketing violations by making them explicit and popping closed questions off the stack. This allowed us to use VPA to recognise the strategies of all third-order terms and hence show observational equivalence is decidable for this fragment. We then showed how to check the inclusion relation of [75], using a product construction and storing P-questions from negative occurrences of var on the stack. This means that our decidability result also holds for the language $\mathrm{IA}_{\mathbf{catch}}$ without the bad-variable constructor.

An obvious direction for future work is to consider RML with non-local control flow. Our decidability results for IA with **catch** relied on the weak bracketing condition. Unfortunately, when answers can justify questions, the visibility condition is no longer sufficient to imply weak bracketing. In fact, when bracketing is dropped from the game-semantic model of RML, weak bracketing is not even preserved by composition of strategies. This difference is reflected in the power of the control operators. Dropping bracketing from the model of IA corresponds to adding the **catch** operator which allows us to "pop the control stack downwards". Removing bracketing from the call-by-value model corresponds to adding **call/cc** (or equivalently Felleisen's $\mathcal{C}$ [37] or the naming operators of $\mu\mathrm{PCF}_v$ [89, 88]) which can additionally "pop the control stack upwards" [61].

Without weak bracketing, there is very little about the semantics which resembles a stack-like behaviour. As such it appears unlikely that the same ideas we used for IA with **catch** can be used to show decidability for a fragment of RML with **call/cc**. We hope it may be possible to show that the O-strict fragment of RML with **call/cc** is decidable. However, without bracketing many type sequents which are O-strict for RML without non-local control are no longer O-strict in the presence of **call/cc**, so the O-strict fragment is much smaller.

Alternatively, we have seen in Chapter 4 that automata with infinite alphabets can be used to capture non-O-strict strategies. It is possible that these might prove useful in deciding fragments of RML with non-local control flow.

The work we have presented so far has concentrated on decidability and complexity results. In the next chapters we move away from theoretical results and present tools we have implemented using game semantics based algorithms.

Chapter 6

Implementation

Thus far we have considered theoretical results. We have identified fragments of languages which have decidable observational equivalence problems. We now turn to implementation. We know that observational equivalence is decidable for both third-order IA and O-strict RML. Further, we know that it is an ExpTime-complete problem in both cases. However, it is unclear whether this bound is reached only in pathological cases or whether in practice it is possible to efficiently equivalence check many interesting programs. In this chapter we discuss our implementations of these algorithms and explore their performance on a number of examples.

6.1 Homer

We first examine Homer [49]. Homer (Higher-order Observational equivalence Model checkER) is an equivalence checker for the third-order fragment of IA. We used F# to implement the result of [81]. When presented with two β-normal forms of IA, Homer constructs VPA which precisely represent their game semantics. This construction is performed inductively over the structure of the terms. The automata are constructed so that the two terms are observationally equivalent if and only if the languages accepted by the automata are equal. We then construct a VPA which recognises the symmetric difference of the languages of the two automata using a product construction. This reduces language equivalence to an emptiness test. The emptiness test is performed using Schwoon's pre* algorithm for reachability in pushdown automata [96]. If the two VPA do not accept the same language, then the reachability check will find a word accepted by one but not the other. This word will be a complete play from the strategy denotation of one of the terms. This acts as a game-semantic counterexample and is essentially a sequence of interactions which one of the terms can have with the context which the other cannot. For the benefit of those

less familiar with the game-semantic model, HOMER also generates an operational-semantic counterexample in the form of a separating context. This is generated from the discovered play using innocent factorisation and finite definability in the style of [6]. When given a term of the correct type, the generated context will terminate if and only if that term can perform the sequence of interactions corresponding to the play.

Since HOMER uses an explicit state representation, large increases in the state space can badly affect the running time. Further, at each stage in the construction it is possible for many unreachable states to be generated. To prevent this, HOMER uses a naïve reachability algorithm to detect states which can be safely deleted. This works by following all transitions out of the initial state, but only following pop-transitions if at least one push-transition on the appropriate stack-symbol has previously been seen. This is very fast and can be performed many times during the construction. It is also (usually) very effective, but it is not guaranteed to delete all unreachable states and does not attempt to identify and merge bisimilar states. Hence, the produced automata are not minimal but experiments showed that this was a good compromise.

We believe HOMER was the first model checker for third-order programs, which makes a fair comparison with existing tools tricky. Two pre-existing game semantics based model checkers are MAGE [12] and GAMECHECKER [35]. These both check only reachability and only at most second-order programs. We had difficulty getting GAMECHECKER to run but compare against MAGE where appropriate (MAGE is supposed to be significantly faster than GAMECHECKER). All performance data refers to running the tools on a quad-core 2.4GHz Intel(R) Xeon(R) with 12GB of memory.

6.1.1 Sorting

As a first example we consider sorting algorithms. Sorting is a difficult problem for a model checker due to the complex interplay between state and control flow. In Figure 6.1 we give a version of bubble-sort. Note that we use the free-identifier x as an input/output-stream. First we read from x to populate the array a, then we sort the array, before finally writing the contents of the array back to x in order. The VPA produced by HOMER for sorting arrays of length two containing three-valued elements is shown in Figure 6.2. For larger arrays the automaton quickly becomes too large to display. The compilation to VPA from IA_3^* only pushes and pops on third-order questions. As this is a first-order example the VPA does not use its stack (i.e. this is a degenerate case where we actually produce a finite automaton). We can see

```
1  x : var ⊢
2  new a[N] in
3  {new i in while !i < N do {a[!i] := !x; i := (!i + 1)}};
4  {
5    new flag in
6    flag := 1;
7    while !flag do{
8      new i in
9      flag := 0;
10     while !i < N−1 do{
11       if !a[!i] > !a[!i + 1] then{
12         new temp in
13         flag := 1;
14         temp := !a[!i] ;
15         a[!i] := !a[!i + 1];
16         a[!i + 1] := !temp
17       }
18       else skip;
19       i := !i + 1
20     }
21   }
22 };
23 {new i in while !i < N do {x:= !a[!i];i := !i + 1}}
```

Figure 6.1: Bubble-sort

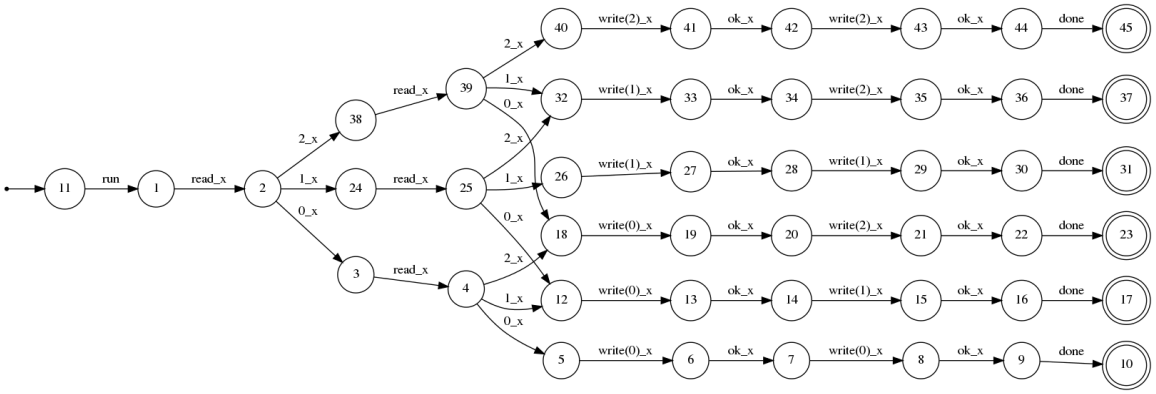


Figure 6.2: VPA produced by HOMER for bubble-sort

n	Time (HOMER)	States (Final)	States (Max)	MAGE (Bubble)	MAGE (Select)
5	1.5s	500	2,700	0.3s	1s
7	9.5s	2,800	19,000	3.3s	10s
10	7min	61,000	470,000	1.75min	5.75min

Table 6.1: Running times for comparing sorting algorithms

that the reads and writes to x are all that is visible to the environment. Hence, if we replace the body of the bubble-sort algorithm with another sorting algorithm, such as select-sort the result is observationally equivalent. The amount of time HOMER takes to perform this check is listed in Table 6.1 for various lengths of lists of three-valued elements. We also list the state space of the final model. This state space is relatively small as the game semantics hides all of the internals of the program and exposes only the aspects which can be detected from outside. Unfortunately, the size of some of the intermediate VPA constructed before reaching the final model are larger (the maximum size is also listed in the table) but are still significantly smaller than the overall state space of the program. For comparison, we also include running times for MAGE in Table 6.1. Since MAGE can only check reachability we cannot perform the same test. Instead, we modify the input program given to MAGE to run the algorithm and then check the resulting array is sorted by asserting that each element in the array is less than or equal to the next element. MAGE then checks that none of these assertions can ever fail. Since MAGE only looks at one program at a time we list running times for its performance on both bubble sort and select sort. As can be seen HOMER's performance is within an order of magnitude of MAGE's which is encouraging considering HOMER is checking a more complex property.

6.1.2 Kierstead Terms

In the sorting example, the VPA we produced were actually finite automata. To get automata which utilise the stack, we consider the Kierstead terms:

$$\lambda f^{(\mathsf{com}\to\mathsf{com})\to\mathsf{com}}.f(\lambda x.f(\lambda y.x)) \text{ and } \lambda f^{(\mathsf{com}\to\mathsf{com})\to\mathsf{com}}.f(\lambda x.f(\lambda y.y)).$$

These terms give rise to the smallest plays for which the location of justification pointers is ambiguous. The automata produced by HOMER are shown in Figure 6.3. The stack is used to keep track of nested calls performed by the context. The justification pointers are encoded by tagging the third-order P-questions. In the first automaton we tag the move labelled `run_1_1_1` with an additional tag of `0` to show that this move refers to f's argument's argument from the outermost of the nested calls to f

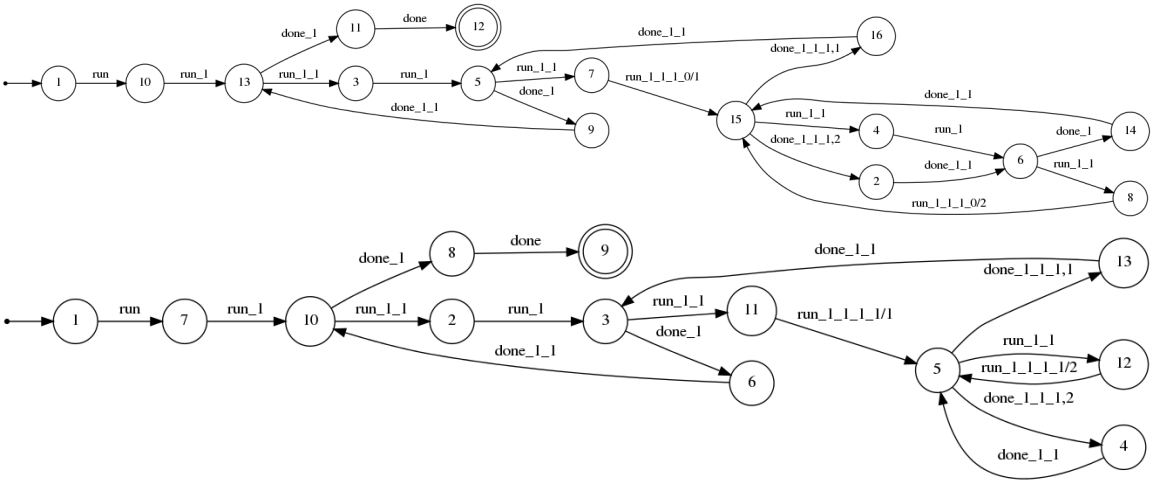


Figure 6.3: Automata for the Kierstead terms

performed by the term. In the second automaton this move is instead tagged with a 1 to show it refers to the argument from the first nested call. It seems intuitive that the two terms should be inequivalent, but it is not at all obvious how to construct a separating context which exhibits their inequivalence. HOMER is able to do this automatically. The separating context generated for these terms is shown (suitably adjusted for readability) in Figure 6.4. The context is generated from the shortest game-semantic play found in the denotation of one but not the other. That is, it is designed so that when a term of the correct type is placed in it, the resulting program will terminate if and only if the supplied term can follow a particular execution path. The variable X is used to force the term to perform specific actions in a specific order, diverging if it does not. By carefully following the execution path we can see that it will terminate when given the second of the Kierstead terms, but diverges when given the first at the point where the term tries to run its argument's argument from the first invocation, deviating from the expected execution path.

6.1.3 No Snapback and Scope Extrusion

Recall the No Snapback and Scope Extrusion examples from Example 2.1 and Example 2.2.

$$p : \mathsf{com} \to \mathsf{com} \vdash \mathbf{new}\, X\, \mathbf{in}\, p(X := 1); \mathbf{if}\, !X = 1\, \mathbf{then}\, \Omega\, \mathbf{else}\, \mathbf{skip} \cong p(\Omega)$$

```
1  (fun G:(((com → com) → com) → com).new X in
2  X:=1;
3  G ( fun z:(com → com).
4     if !X = 1 then
5     (
6       X:=2;
7       z Ω;
8       if !X = 5 then X:=6 else Ω
9     )
10    else if !X = 2 then
11    (
12      X:=3;
13      z (if !X = 3 then X:=4 else Ω);
14      if !X = 4 then X:=5 else Ω
15    )
16    else
17      Ω
18 );
19 if !X = 6 then X:=7 else Ω
20 )([−])
```

Figure 6.4: Separating context for Kierstead terms

$$M_1 \equiv \lambda F^{(\mathsf{exp}\to\mathsf{exp})\to\mathsf{exp}}.\mathbf{new}\, x := 0\, \mathbf{in}\, F\,(\lambda y^{\mathsf{exp}}.\mathbf{if}\, !x = 0\, \mathbf{then}\, x := y\, \mathbf{else}\, x := y - 1;\, !x)$$

$$M_2 \equiv \lambda F^{(\mathsf{exp}\to\mathsf{exp})\to\mathsf{exp}}.F\,(\lambda y^{\mathsf{exp}}.\mathbf{new}\, x := 0\, \mathbf{in}\, \mathbf{if}\, !x = 0\, \mathbf{then}\, x := y\, \mathbf{else}\, x := y - 1;\, !x)$$

$$M_3 \equiv \lambda F^{(\mathsf{exp}\to\mathsf{exp})\to\mathsf{exp}}.F\,(\lambda y^{\mathsf{exp}}.y)$$

$$M_1 \ncong M_2 \cong M_3$$

All these terms are in the third-order fragment of IA. These (in)equivalences are tricky to reason about by hand. However, for HOMER they are straightforward and can be checked in less than a second as seen in Table 6.2. These examples are inherently equivalence checks and the scope extrusion examples are third-order so we cannot give running times for MAGE.

6.1.4 Model-Checking Regular Properties

Another feature of HOMER is the ability to check whether a program satisfies some property which can be described as a regular expression. Consider the property

	Time (HOMER)	States (Final)	States (Max)
No Snapback	400ms	9	30
Scope Extrusion (Inequivalence)	600ms	92	1,400
Scope Extrusion (Equivalence)	500ms	40	420

Table 6.2: Running times for no snapback and scope extrusion examples

"Whenever p is run, it is given the value of X as its argument and whenever it terminates its return value is immediately written to X". We can describe this property by the regular expression

$$(\sum_i q\_1\_p \;\; read\_X \; i\_X \;\; i\_1\_p + \sum_i i\_p \;\; write(i)\_X + Y)^*$$

where Y is the set of all moves except q_1_p and i_p. We can then check whether a term satisfies this by checking if the language of the VPA-translate of the term is included in the language of the regular expression. For example, **skip** satisfies it trivially. More interestingly, $X := (p \;\; !X)$ and $X := (X := (p \;\; !X)\,;\, (p \;\; !X))$ both satisfy it but $X := ((p \;\; !X) + 1)$ does not.

6.2 HECTOR

We also have an F# implementation of the decidability result for O-strict RML. We call this tool HECTOR (Higher-order Equivalence Checker for Terms of O-strict RML) [48]. Following the example of MAGE, we decided to follow an on-the-fly model checking approach. That is, rather than building up the entire model and then exploring it, we construct our model as we explore it. In our case, given two programs we need to build an automaton that recognises the symmetric difference of their game-semantic denotations. We initially build up this automaton as a function which maps each state to the set of transitions out of that state. This does not actually evaluate and construct the transitions until it is called by the exploration function. The exploration algorithm then needs to determine whether the language accepted by the automaton is empty or not. If a counterexample is found, the exploration can be terminated without having fully constructed the automaton. Emptiness for pushdown automata is a non-trivial problem. Whereas MAGE, which deals only with finite automata, can simply ascertain whether there is a path from the initial state to the final state in the transition graph, HECTOR needs to take into account stack actions. On-the-fly reachability for recursive state machines (which are equivalent to pushdown automata) was studied in [10] and so we follow their approach. The algorithm, shown

```
1   REACHABILITY(s, s_en, γ)
2     Visited.add(s, s_en, γ)
3     if s ∈ Target then
4       print 'Target Reached'
5       break
6     for e ∈ Transitions(s) do
7       if e is a γ-pop transition s -a,γ-> s_ret then
8         VisitedReturns[s_en, γ].add(s_ret)
9         for (s', s'_en, γ') ∈ VisitedCalls[s_en, γ] do
10          if (s_ret, s'_en, γ') ∉ Visited then
11            REACHABILITY(s_ret, s'_en, γ')
12      else if e is a noop-transition s -a-> s' then
13        if (s', s_en, γ) ∉ Visited then
14          REACHABILITY(s', s_en, γ)
15      else if e is a push transition s -a/γ'-> s'_en then
16        VisitedCalls[s'_en, γ'].add(s, s_en, γ)
17        if (s'_en, s'_en, γ') ∉ Visited then
18          REACHABILITY(s'_en, s'_en, γ')
19        else
20          for s_ret ∈ VisitedReturns[s'_en, γ'] do
21            if (s_ret, s_en, γ) ∉ Visited then
22              REACHABILITY(s_ret, s_en, γ)
```

At each point s is the state we are exploring, γ is the symbol at the top of the stack and s_{en} is the state which was entered as γ was pushed. These last two are needed to add summary edges, as we maintain the invariant that s is reachable from s_{en} using only internal and summary edges.

Figure 6.5: On-the-fly reachability for pushdown automata

in Figure 6.5, relies on summary edges which summarise the transitions the automaton can make between when a symbol is pushed on to the stack and when it is popped off. The exploration essentially proceeds as a depth-first search, recording push- and pop-sites so that the additional summary edges can be added when two matching transitions are found.

6.2.1 Sorting

We again start out by considering sorting algorithms. As before we can compare whether two different sorting algorithms are observationally equivalent. The running times for HECTOR compared to HOMER are shown in Table 6.3. It can be seen that HECTOR is outperformed on this example by HOMER. As this is an example of an equivalence, it is possible that the slow performance of HECTOR is due to

n	HECTOR to Compare	States	HOMER	Final States	Max States
5	4s	720	1.5s	500	2,700
7	1.5min	4,900	10s	2,800	19,000
10	2hours	120,000	7.5min	61,000	470,000
With A Comparison Function					
5	220ms	100	2.25min	74,000	74,000
7	225ms	130	Time Out	Time Out	Time Out
10	300ms	190	Time Out	Time Out	Time Out
15	400ms	280	Time Out	Time Out	Time Out

Table 6.3: Time to compare sorting algorithms

the on-the-fly approach being slower when the entire model has to be constructed. However, we suspect that it is actually due to the added complications of the call-by-value model. The constructions HECTOR has to perform require more work and the resulting automata use their stacks more often (as this is a first-order example the automata generated by HOMER do not use their stacks at all, whereas those HECTOR produces do so on almost every transition). The added stack actions make the exploration algorithm slower.

We can also consider a version of the sorting algorithms which are parameterised by a comparison function *compare* : $\mathsf{int} \to \mathsf{int} \to \mathsf{int}$. Instead of using the less-than ordering, the programs can make a call to *compare*. In this case a malicious context could pass in a comparison function which does not act as a total order and can use this function to gain more information about the internals of the algorithm. Hence, the two programs are no longer equivalent. Due to the added size of the model when parameterised in this manner, HOMER runs out of memory for lists of length 7. On the other hand, due to the on-the-fly approach HECTOR finds the counterexample almost immediately and so can terminate early without having to construct the entire model. The running times are also included in Table 6.3.

6.2.2 Kierstead Terms

We can also revisit the Kierstead terms. Previously we considered the terms

$$\lambda f^{(\mathsf{com}\to\mathsf{com})\to\mathsf{com}}.f(\lambda x.f(\lambda y.x)) \text{ and } \lambda f^{(\mathsf{com}\to\mathsf{com})\to\mathsf{com}}.f(\lambda x.f(\lambda y.y)).$$

These generalise into a family of call-by-name terms

$$K_{n,i} := \lambda f^{(\mathsf{com}\to\mathsf{com})\to\mathsf{com}}.f(\lambda x_1.f(\lambda x_2.\ldots f(\lambda x_{n-1}.f(\lambda x_n\,.\,x_i)))).$$

n	HECTOR to Compare	States	HOMER	Final States	Max States
10	120ms	150	1s	74	1,400
25	140ms	370	6s	190	4,100
50	180ms	580	22s	360	7,200
100	530ms	1,600	2min	800	18,000
200	2min	37,000	7min	1,300	42,000

Table 6.4: Time to compare Kierstead terms

For $n, i \neq m, j$ we have that $K_{n,i} \not\cong K_{m,j}$. Under call-by-value we can define the Kierstead terms as

$$K_{n,i} \equiv f : ((\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \to \mathsf{unit} \vdash f(\lambda x_1.f(\lambda x_2. \ldots f(\lambda x_n.x_i()) \ldots)).$$

The running times for checking that $K_{n,i} \not\cong K_{n,j}$ for various values of n and $i \neq j$ are shown in Table 6.4. As this is an inequivalence, the on-the-fly approach of HECTOR allows it to significantly out-perform HOMER.

6.2.3 "Tricky" Examples

Several examples in the literature are known to be challenging to verify. Recall the terms below from Example 3.1 and Example 3.2.

(i) $\mathbf{let}\, c = \mathbf{ref}\, 0\, \mathbf{in}\, \lambda f^{\mathsf{unit}\to\mathsf{unit}}.(c := 1; f(); !c) \;\cong\; \lambda f^{\mathsf{unit}\to\mathsf{unit}}.(f(); 1)$

(ii) $\mathbf{let}\, c = \mathbf{ref}\, 0\, \mathbf{in}\, \lambda f^{\mathsf{unit}\to\mathsf{unit}}.(c := 0; f(); c := 1; f(); !c) \;\cong\; \lambda f^{\mathsf{unit}\to\mathsf{unit}}.(f(); f(); 1)$

(iii)
$$\begin{array}{l} p : (\mathsf{unit} \to \mathsf{int}) \to \mathsf{unit} \vdash \\ \quad \mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in} \\ \quad p(\lambda z.x := 1; 0); \\ \quad \mathbf{let}\, y = !x\, \mathbf{in}\, \mathbf{if}\, y = 0\, \mathbf{then}\, ()\, \mathbf{else}\, \Omega \end{array} \quad \cong \quad p(\lambda z.\Omega)$$

(iv) $\mathbf{let}\, a \;=\; \mathbf{ref}\, 0\, \mathbf{in}\, \mathbf{let}\, r \;=\; \mathbf{ref}\, 0\, \mathbf{in}\, \lambda f.(r := !r + 1; a := f(!r); r := !r - 1; !a) \quad \not\cong \lambda f.f(1)$

As previously mentioned, the three equivalences are known to be extremely tricky to prove using methods based on logical relations [90, 9, 17] and the inequivalence requires a rather delicate context to separate the terms [101]. However, all of these examples are in the O-strict fragment and HECTOR can easily handle them in milliseconds as seen in Table 6.5.

Example	Time to Compare	State Space
(i) [90]	180ms	67
(ii) [9]	130ms	230
(iii) [17]	160ms	16
(iv) [101]	150ms	57

Table 6.5: Running times for tricky examples

6.2.4 Order

Inspired by our previous examples we can consider a series of small terms which illustrate the increasing complexity as we increase the order of functions. Starting at order 0 we have that

$$\vdash \mathbf{let}\ \mathit{count} = \mathbf{ref}\,0\,\mathbf{in}\,(\mathit{count} := !\mathit{count} + 1; \mathbf{if}\,!\mathit{count} < 10\,\mathbf{then}\,()\,\mathbf{else}\,\Omega)$$

is equivalent to $\vdash ()$. This is because an order 0 type can only be run once and so the value in *count* can never be increased beyond 1.

If we move to order 1 and consider similar terms the equivalence no longer holds. The term

$$\vdash \mathbf{let}\ \mathit{count} = \mathbf{ref}\,0\,\mathbf{in}\,(\lambda c : \mathsf{unit}.\mathit{count} := !\mathit{count} + 1; \mathbf{if}\,!\mathit{count} < 10\,\mathbf{then}\,c\,\mathbf{else}\,\Omega)$$

is not equivalent to the term $\lambda c : \mathsf{unit}.c$. The λ-abstraction allows the body to be called multiple times which can increase *count* above 10. However, at order 1 these repeated calls must be performed sequentially; in the absence of recursion (and higher-order identifiers) we cannot have multiple calls to the same first-order function open at once. Hence, the following terms are equal

$$\begin{array}{l}\vdash \mathbf{let}\ \mathit{count} = \mathbf{ref}\,0\,\mathbf{in}\\ \lambda c : \mathsf{unit}.(\mathit{count} := !\mathit{count} + 1; (\mathbf{if}\,!\mathit{count} < 10\,\mathbf{then}\,c\,\mathbf{else}\,\Omega); \mathit{count} := !\mathit{count} - 1)\end{array}$$

and $\lambda c : \mathsf{unit}.c$. Since *count* is incremented after the function is called and decremented at the end of the function body, its value can never grow above 1.

However, if we increase the order again and consider order 2 functions then this property that there can only be one copy of a function open at once fails. That is, the following terms are inequivalent.

$$\begin{array}{l}\vdash \mathbf{let}\ \mathit{count} = \mathbf{ref}\,0\,\mathbf{in}\\ \lambda f : \mathsf{unit} \to \mathsf{unit}.\\ (\mathit{count} := !\mathit{count} + 1; (\mathbf{if}\,!\mathit{count} < 10\,\mathbf{then}\,f()\,\mathbf{else}\,\Omega); \mathit{count} := !\mathit{count} - 1)\end{array}$$

and $\lambda f : \mathsf{unit} \to \mathsf{unit}.f()$. This is similar to Stark's tricky example. The identifier f can be instantiated with a function which makes further calls to the term. In particular, they are separated by an application of the term

$$\lambda F : (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}.$$
$$F(\lambda x_1.F(\lambda x_2.F(\lambda x_3.F(\lambda x_4.F(\lambda x_5.F(\lambda x_6.F(\lambda x_7.F(\lambda x_8.F(\lambda x_9.F(\lambda x_{10}.()))))))))))).$$

This causes 10 copies of the function F (which will be bound to one of our terms) to be open at once. When provided with the first term this causes *count* to reach value 10 leading to divergence. HECTOR can easily verify all of these equivalences and inequivalences, taking around 150ms for each. These examples illustrate the increasing power contexts gain as the order of our terms increase. This shows up in the types of automata we need to capture their game semantics. The order 0 terms we considered could not be run more than once and so finite automata without any loops suffice to capture these. At order 1 we can iterate the function body but cannot contain any nesting. This can still be captured using finite automata but the automata will need to contain cycles (or if using regular expressions we will need to use Kleene star). Finally, for our second-order example regular expressions no longer provide sufficient expressive power as the context can perform nested calls to our function. To represent this we need to use pushdown automata so that we can use the stack to match calls to returns.

6.2.5 Reachability

In addition to checking equivalences, HECTOR can also perform reachability tests. To do this we add assertions to terms. The statement **assert**(b) is treated as an abbreviation for **if** b **then** () **else** *fail*() where *fail* is a special identifier only able to be introduced through assertions. HECTOR then checks if the strategy denoting a term contains any plays with *fail*-moves. This will be the case if there exists a context the term could be placed in which would cause any of the assertions to fail.

In Figure 6.6 is an implementation for a stack data structure using a fixed length array. The function to push an element on to the stack first asserts that the array is not full. Similarly, before popping we assert that the stack is non-empty. We also have an invariant function which checks that if we push an element on to the stack and then immediately pop it off, we do indeed get the same element back. The undefined function VERIFY is used to allow the environment to call these functions in any way it desires. Clearly this will allow the context to pop the empty stack or push to a full stack causing our assertions to fail. By removing all but one of the assertions we can

```
1  VERIFY : (int → unit) → (unit → int) → (int → unit) → unit ⊢
2  let a = [| 0 , 0 , 0 , 0 , 0 |] in
3  let size = 5 in
4  let current = ref 0 in
5  let isEmpty = fun x . !current = 0 in
6  let isFull = fun x . !current = size in
7  let push =
8    fun x .
9      assert (isFull() = 0);
10     a[!current] := x;
11     current := !current + 1
12   in
13 let pop =
14   fun x .
15     assert (isEmpty () = 0);
16     current := !current − 1;
17     !a[!current]
18   in
19 let invariant =
20   fun x . (push x ; assert (pop () = x)) in
21 VERIFY push pop invariant
```

Figure 6.6: Code for a stack data structure

check for overflow, underflow and violation of the invariant separately (although since attempts to access non-existing array elements cause divergence, the invariant will always hold). Running times for a variety of array lengths are shown in Table 6.6. We compare running times against those for MAGE on an similar encoding of a stack. As this is exactly the type of example MAGE is designed to handle it should be no surprise that HECTOR is outperformed. However, the performance, at least when checking for overflow, is still encouraging. Since we pass VERIFY the push function before the pop and invariant functions, the model checkers' depth first searches both explore the uses of push first. This means that the possible overflow is detected relatively quickly. Unfortunately, the downside of this is that when we check for underflow, the search explores the wrong part of the model. Even though the shortest possible counterexample is in fact very short, as we are using a depth-first search we do not find it. Consequently, HECTOR struggles to find the underflow error. When checking the invariant holds both model checkers struggle as this is an example where the entire state space needs to be explored.

Interestingly, for the overflow and underflow errors in this example, the actual values of the data stored on the stack are almost irrelevant. MAGE uses data abstraction

Size of Array	HECTOR	State Space	MAGE
Overflow			
5	175ms	36	4ms
10	190ms	61	5ms
20	260ms	130	9ms
40	700ms	250	19ms
80	4.2s	490	38ms
Underflow			
5	1.5s	6,000	6ms
10	Time Out	Time Out	9ms
Invariant			
5	Time Out	Time Out	21s
10	Time Out	Time Out	Time Out
Overflow 1024			
5	240ms	6,200	4ms
10	330ms	11,000	6ms
20	700ms	22,000	9ms
40	2s	42,000	18ms
80	10.5s	83,000	37ms

Table 6.6: Running times on the stack example

to take advantage of this and so its performance is almost unaffected by the size of the data. HECTOR does not use any abstraction but even so can still handle large integers. Most of the entries in Table 6.6 were carried out with three-valued integers, but the final section gives running times for overflow checking stacks of 1024-valued integers.

This example also illustrates a downside to the way HECTOR is implemented. HECTOR performs its model checking on-the-fly, but this refers to the construction and exploration of the automata. Before this construction is performed, the input is parsed and then converted into canonical form explicitly. In most examples this is not a problem as the model checking is still the most expensive part of the computation. However, when checking for overflow in the stack example, the normal form is very large but the counterexample is very small and HECTOR spends more time preprocessing the term than actually model checking. In examples like this it would perhaps be better to work with the term directly (or convert to canonical form on-the-fly) rather than explicitly performing the conversion before model checking.

6.3 Summary

In this chapter we have presented our equivalence checkers HOMER and HECTOR for third-order IA and O-strict RML respectively. These both follow an explicit state representation for the VPA translates of terms. As such, on large examples they struggle to deal with the state space explosion problem. However, despite this they are both still able to handle many interesting and challenging examples. HECTOR uses an on-the-fly model checking approach which allows it to perform particularly well on inequivalences as it can terminate as soon as a counterexample is found without having to construct the whole model.

As there are no other existing equivalence checkers which can handle such higher-order languages, a fair comparison with existing tools is tricky. We chose to compare against MAGE as it is a similar tool, with a similar input language even though it cannot handle as higher-order programs and checks only for reachability. Since MAGE checks a simpler problem (and also takes advantage of CEGAR) it is not surprising that on most examples MAGE runs quicker, but the fact that HOMER and HECTOR's running times are often within an order of magnitude is encouraging.

In the next chapter we move away from the explicit state representation and consider a method for representing strategies based on symbolic execution.

Chapter 7

Symbolically Executed Automata

In the previous chapter we discussed our model checkers HOMER and HECTOR which determine observational equivalence of third-order IA and O-strict RML respectively. Although HECTOR follows an on-the-fly approach, both tools work by constructing an explicit representation of a VPA which recognises the strategy denotation of a term. One downside of this approach is that the size of the alphabets can be very large as it can be proportional to the size of the integer-type. Large alphabets usually entail large automata and slow model checking. For example, consider the two versions of the IA identity function $\lambda x : \mathsf{exp}.x$ and $\lambda x : \mathsf{exp}.\mathbf{new}\, X\, \mathbf{in}\, X := x; !X$. These two functions are equivalent since the outside environment cannot detect the use of the local variable X. Complete plays from their denotation will have the form $q \cdot q_1 \cdot \sum_i i_1 \cdot i$. This seems very simple but the sum is over all values in exp. If we increase the maximum integer value, then the size of both the alphabet and state space of the automaton required to represent this strategy grow proportionally (so exponentially in the size of the integers). Even on this simple example, HOMER takes 5mins if 10-bit integers are considered.

To overcome the state space explosion problem, model checking methods based on symbolic representations have been proposed [68]. These often involve using BDDs [24] to encode a transitions system [25] or (if BDDs no longer scale sufficiently) using SAT- or SMT solvers and bounded model checking [18]. Here we focus on symbolic execution (or symbolic simulation) which has proved successful in hardware verification [23]. It involves executing a program (or circuit) using symbolic formulas instead of concrete values. The formulas describe the possible values which can be associated with a variable or program point. Recently, symbolic execution has been used for equivalence checking high-level specifications against circuit descriptions [50], equivalence of C-like circuit specifications [65] and equivalence of embedded software

in assembly-language [33]. As symbolic execution is known to be an effective technique for equivalence checking, it is natural to investigate whether it can be adapted to the game-semantic setting.

Symbolic execution is based on following the execution path of a program. Game semantics, on the other hand, completely hides the actual computation, exposing only those actions observable from the outside. At first glance then, we may conclude that the two ideas are incompatible. However, recall our previous example of the identity function. We said that the plays from the identity strategy and its observational equivalence class have the form $q \cdot q_1 \cdot i_1 \cdot i$. Our two programs have different execution paths, since the second involves the allocation, assignment and dereferencing of a local variable, but both result in the same plays. In fact, any term equivalent to the identity will have the same plays and the exact execution paths that give rise to these plays will be hidden. However, in a certain sense, this sequence of visible actions is still executed. Intuitively, the move i_1 is an input to the execution and the second i, which must match the first, is an output. As we previously noted, i is allowed to range over the exp data type, so if we represent the automaton for this strategy explicitly then the number of states will grow exponentially with the length of the integers we allow. Since the relationship between the moves is so simple this seems rather excessive. All we want is an automaton similar to the one in Figure 7.1 with five states. Here we use the notation of question marks for inputs and exclamation marks for outputs. This is obviously much more compact and, assuming the range

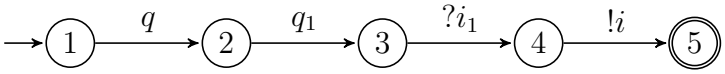


Figure 7.1: A compact representation of the identity strategy

of possible inputs is clear, precisely represents the desired automaton. By viewing all O-moves as inputs and all P-moves as outputs we can extend this idea beyond this single example. In this chapter we propose a formal definition for such automata, discuss how to compile IA terms to such automata via game semantics and how to model check them. We also test our implementation on a number of examples and evaluate its performance.

7.1 Definition of Symbolically Executed Automata

We start by defining a class of automata which aim to capture this intuition. We assume that our finite alphabet Σ is given and that we have a countable set of variables *Var* which range over Σ, a set of expressions *Expr* over *Var* which denote members of Σ and a set of formulas *Form* over *Var* and *Expr* (sometimes referred to as predicates). We assume that for an expression E and valuation function V mapping the (free) variables of E to members of Σ, $eval_V(E)$ returns the member of Σ represented by E under valuation V.

In our intended usage, Σ will be the set of moves from our arena, *Expr* will contain all the arithmetic operators of IA and *Form* will contain all the comparison operators of IA as well as the usual boolean connectives. This contains some implicit typing, for example the expression $q + 1$ (where q is the question-move corresponding to asking for the value of an expression) does not make sense. Typing information can be encoded into the formulas used but we will not make this explicit. Similarly, while moves are often tagged to make it explicit which side of a disjoint sum they come from, we will not always explicitly detag them when using them in expressions. For example, $1_1 + 3_2$ should be taken to have the (untagged) value 4.

Definition 7.1. A *Symbolically Executed Automaton* (SEA) is a tuple $\langle Q, q_0, F, \delta \rangle$ where:

- Q is a finite set of states.
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is the set of final (or accepting) states.
- $\delta : (Q \times (\mathcal{P}_f(Var) \times Form) \times Q) + (Q \times (Expr \times Form) \times Q)$ is the transition function.

A transition $q_1 \xrightarrow{(X,\psi)} q_2$ from $Q \times (\mathcal{P}_f(Var) \times Form) \times Q$, will be referred to as an *input* (or an X-input, or x-input for any $x \in X$). We will often denote this $q_1 \xrightarrow{?X\,:\,\psi} q_2$, or if X is the singleton set $\{x\}$, $q_1 \xrightarrow{?x\,:\,\psi} q_2$. Conversely, a transition $q_1 \xrightarrow{(E,\psi)} q_2$ from $Q \times (Expr \times Form) \times Q$, will be referred to as an *output*. We will often denote this $q_1 \xrightarrow{!E\,:\,\psi} q_2$, sometimes omitting the !. We will use $q_1 \xrightarrow{T\,:\,\psi} q_2$ to range over transitions which could be either an input or an output.

We let a *path* be a sequence of transitions $q_0 \xrightarrow{T_0\,:\,\psi_0} q_1 \xrightarrow{T_1\,:\,\psi_1} \ldots \xrightarrow{T_{n-1}\,:\,\psi_{n-1}} q_n$ starting at the initial state, such that each transition is in δ. A path is *well-formed*

if whenever a variable x occurs (free) in some ψ_i or output expression E_i, there is an x-input transition earlier in the sequence ($q_k \xrightarrow{?X_k : \psi_k} q_{k+1}$, $k \leq i$, $x \in X$). Note that it is intended that this does allow ψ_i to contain variables from X_i if the ith transition is an input $q_i \xrightarrow{?X_i : \psi_i} q_{i+1}$. We say an automaton is well-formed if every possible path is well-formed. From now on we assume all our automata are well-formed.

Given a word w (or path π), let w^i (π^i) be the prefix of w (π) of length i. Also, let w_i be the i-th letter of w (counting from zero). So if $w = a \cdot b \cdot c$ then $w^2 = a \cdot b$ and $w_2 = c$. Given a path π, a word w of the same length and a variable x (of which π contains at least one x-input transition) let

$$val_{\pi,w}(x) = w_i \text{ where } i \text{ is the index of the last } x\text{-input transition in } \pi.$$

Example 7.1. Recall the automaton from Figure 7.1. This can be made to fit our definition by making some of the expressions and predicates more explicit as in Figure 7.2. This is well-formed, since the only variables occurring in predicates are

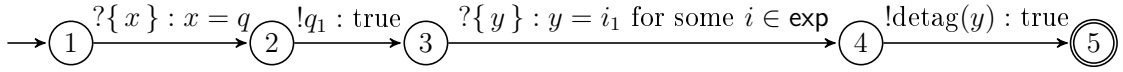


Figure 7.2: The automaton from Figure 7.1 with explicit formulas and expressions

on input transitions for that variable and the only variable on an output transition is y, which clearly must occur after the y-input transition.

If π is the (unique) path from state 1 to state 4 and w is the word $q \cdot q_1 \cdot 2_1$, then $val_{\pi,w}(x) = q$ and $val_{\pi,w}(y) = 2_1$.

We say a path $\pi = q_0 \xrightarrow{T_0 : \psi_0} q_1 \xrightarrow{T_1 : \psi_1} \ldots \xrightarrow{T_{n-1} : \psi_{n-1}} q_n$ is a *run* on word w if for each output transition $q_i \xrightarrow{E_i : \psi_i} q_{i+1}$, we have $eval_{val_{\pi^i,w^i}}(E_i) = w_i$. A run is *valid* if $\forall i : 0 \leq i < n : val_{\pi^{i+1},w^{i+1}} \models \psi_i$. A run is *accepting* if it is valid and ends in a final state. For an automaton M, $\mathcal{L}(M) = \{w \mid M \text{ has an accepting run on } w\}$.

Intuitively, each input transition binds all its variables to whichever input is provided. Output transitions can only consume a particular letter, which may depend on the latest values bound to each variable. Both input and output transitions can only be taken if the guard on that transition holds. This may also depend on the last value bound to each variable.

Example 7.2. In the automaton from Example 7.1, the (unique) path from state 1 to state 5 is an accepting run on the word $q \cdot q_1 \cdot 2_1 \cdot 2$. In fact, this path will be an accepting run on any word of the form $q \cdot q_1 \cdot i_1 \cdot i$ for some i, which is exactly what we wanted.

Example 7.3. In Figure 7.3 we show an SEA for comparing two integers. The automaton reads two integers x and y and then writes their values into z in non-decreasing order. This time we have omitted the implicit typing and untagging, as well as predicates for inputs and outputs that can only take one value.

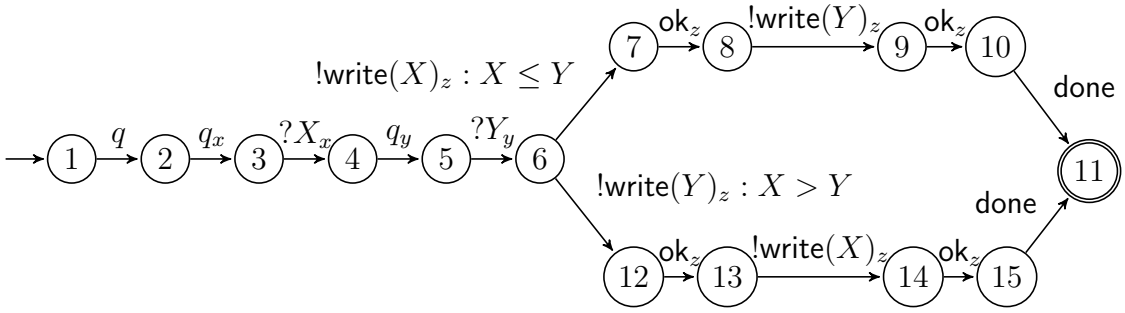


Figure 7.3: An SEA for comparing two integers

Translation to Finite Automata SEA are designed to be a compact representation of finite automata. Given an SEA $\mathcal{A}_1 = \langle Q, q_0, F, \delta \rangle$, we can construct an equivalent finite automaton $\mathcal{A}_2$ which accepts the same language. If the finite set of variables occurring in transitions of $\mathcal{A}_1$ is $Var_{\mathcal{A}_1}$ then let $Val_{\mathcal{A}_1}$ be the finite set of partial valuation functions $Var_{\mathcal{A}_1} \rightharpoonup \Sigma$. We construct $\mathcal{A}_2$ as follows:

- The set of states is $Q \times Val_{\mathcal{A}_1}$.
- The initial state is $(q_0, \emptyset)$.
- The set of final states is $\{ (f, V) \mid f \in F \}$.
- There is a transition $(q, V) \xrightarrow{a} (q', V')$ if
  - $q \overset{?X\,:\,\psi}{\rightarrow} q' \in \delta$ such that $V' = V[X \mapsto a]$ and $V' \models \psi$,
  - or $q \overset{!E\,:\,\psi}{\rightarrow} q' \in \delta$ such that $V = V'$, $eval_V(E) = a$ and $V \models \psi$.

The constructed finite automaton $\mathcal{A}_2$ will accept exactly the same language as the SEA $\mathcal{A}_1$. By keeping track of the valuation function, the runs of $\mathcal{A}_2$ correspond exactly to the valid runs of $\mathcal{A}_1$.

7.2 Game Semantics to SEA

We now consider how to translate IA terms into SEA. We consider only the first-order imperative fragment of IA. This simplifies the semantics while still being relatively expressive, so is a suitable fragment for testing our ideas on. This fragment is contained in the language given a regular language semantics in [42]. We will insist that all O-transitions are input-transitions while all P-transitions are outputs. Most of the constructions from [42] are simple to adapt to construct SEA. For example, free identifiers are copycat strategies and so every P-output will just output the immediately preceding O-input. We just need to constrain the possible inputs to obey the rules of the game. The majority of the imperative constructs just involve combining the automata representing subterms, usually redirecting and hiding initial and final moves. Care has to be taken with the hidden formulas and expressions, but it is not too different from the explicit case.

Example 7.4. As an illustrative example, consider the case of $\mathbf{while}\, M\, \mathbf{do}\, N$. We are given SEA $\mathcal{A}_M$ and $\mathcal{A}_N$ which recognise $[\![M]\!]$ and $[\![N]\!]$ respectively. Note that as these automata will be deterministic and the strategies $[\![M]\!]$ and $[\![N]\!]$ both have unique initial moves, $\mathcal{A}_M$ and $\mathcal{A}_N$ must have only one transition which can be taken from the initial state and its guard will be equivalent to *true*. We refer to the states reached by following these initial transitions as q_{1M} and q_{1N} respectively.

To construct an SEA $\mathcal{A}_{\mathbf{while}}$ to recognise $[\![\mathbf{while}\, M\, \mathbf{do}\, N]\!]$ we take as the set of states the disjoint union of the states from $\mathcal{A}_M$ and $\mathcal{A}_N$. The initial and final states will be those of $\mathcal{A}_M$.

The transitions will be as follows:

- The initial transition of $\mathcal{A}_M$ is relabelled with run. That is, we have $q_{0M} \xrightarrow{\mathsf{run} : \mathit{true}} q_{1M}$.
- All transitions which are not initial or final are preserved. That is, if $s_1 \xrightarrow{T : \psi} s_2$ in either $\mathcal{A}_M$ or $\mathcal{A}_N$ where s_1 is not initial and s_2 is not final in the appropriate automaton, then we keep this transition unchanged in $\mathcal{A}_{\mathbf{while}}$.
- If $s_1 \xrightarrow{!E : \psi} f$ in $\mathcal{A}_M$ where f is final, we have two transitions based on whether the value of E requires us to run the body of the loop or terminate. We use the guards to check the value of E. We have $s_1 \xrightarrow{\mathsf{done} : E{=}0 \wedge \psi} f$ and $s_1 \xrightarrow{\epsilon : E{>}0 \wedge \psi} q_{1N}$.

- Lastly, final transitions in the body-automaton are redirected back to the guard automaton. If $s_1 \xrightarrow{!E : \psi} f$ in $\mathcal{A}_N$ where f is final (E must evaluate to done in any valid run) then $s_1 \xrightarrow{\epsilon : \psi} q_{1M}$.

This construction involves ϵ-transitions which are not part of our definition of SEA. This is not a problem though, as we can easily remove such transitions as the final step of the construction. For every sequence of transitions

$$s_1 \xrightarrow{\epsilon : \psi_1} s_2 \xrightarrow{\epsilon : \psi_2} \dots \xrightarrow{\epsilon : \psi_n} s_{n+1} \xrightarrow{!E : \psi_{n+1}} s_{n+2}$$

where each s_i is distinct (and the final transition is not an ϵ-transition), we add a new transition $s_1 \xrightarrow{!E : \bigwedge_i \psi_i} s_{n+1}$. We can then safely remove all ϵ-transitions.

Example 7.5. We now give a concrete example of the above construction. Consider the IA term $x : \mathsf{var} \vdash \mathbf{while}\,!x < 4\,\mathbf{do}\,x :=!x + 2$. Note that we are not wrapping x in a **new**-block so its behaviour is not constrained. The automata for the guard and body of the loop (before they are combined by the **while** construction) are shown in Figure 7.4. Applying our construction gives the SEA in Figure 7.5. We can then

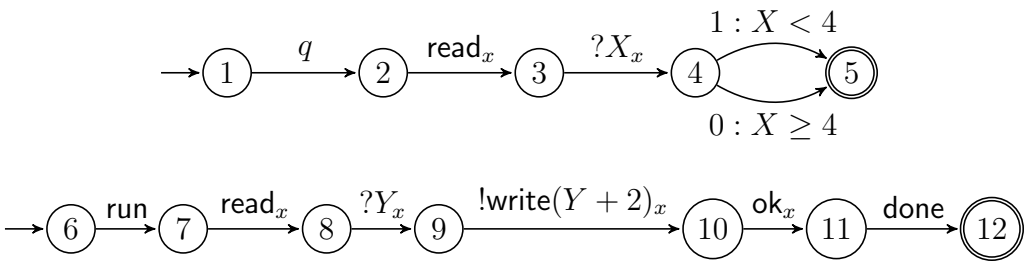


Figure 7.4: SEA for the guard and body of the while-loop

compress the ϵ-transitions as outlined above. Additionally, we can clearly remove transitions with guards which are always unsatisfiable (such as $1 = 0$) and similarly simplify guards containing conjuncts which are always true. The final SEA is shown in Figure 7.6 and has the form we might have expected.

7.2.1 Local Variable Blocks

The above ideas essentially allow us to model the control flow of the program in a reasonably simple manner. Unfortunately, things become more complicated when we try to model the state as well. To construct an automaton for $[\![\Gamma \vdash \mathbf{new}\,X\,\mathbf{in}\,M]\!]$ we need to take the SEA for $[\![\Gamma, X \vdash M]\!]$, restrict it to "good-variable" behaviour and hide

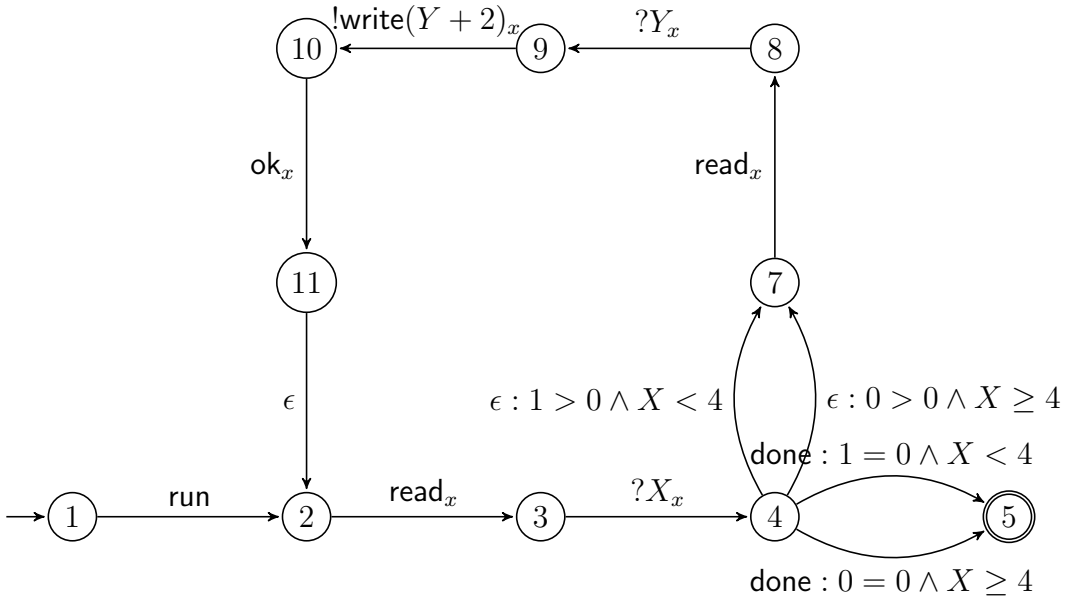


Figure 7.5: SEA for the while-loop

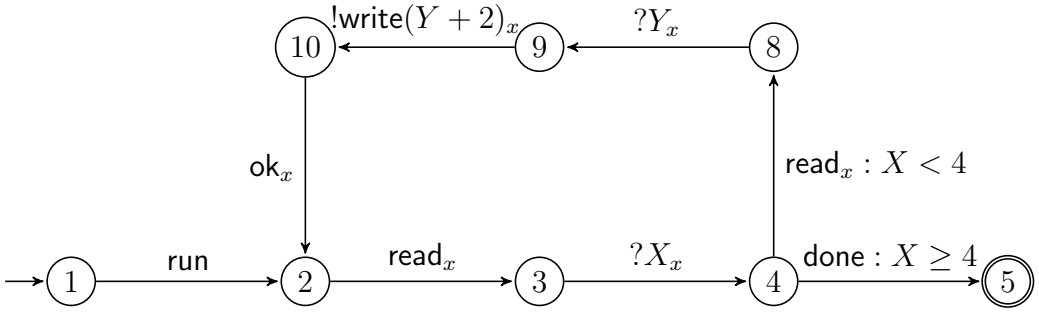


Figure 7.6: SEA for the while-loop with ϵ-transitions removed and guards simplified

all X-transitions. When representing this with finite automata rather than SEA, we take a copy of the automaton for $[\![\Gamma, X \vdash M]\!]$ for each value which can be written to X. We then redirect write-moves so that $\mathsf{write}(i)_X$ goes to the corresponding state in the ith copy and restrict the answers to read_Xs such that in the ith copy the answer is always i_X. Having done this we turn all the $[\![\mathsf{var}_X]\!]$-moves into ϵ-moves. These ϵ-transitions can be compressed out by adding them onto the next visible transition.

We can attempt to do the same thing in the symbolic case. In our SEA, instead of having moves of the form $\mathsf{write}(0)_X$, we may have transitions on $\mathsf{write}(E)_X$ where E is some expression. Instead of creating a new copy for each possible value the variable could take, we create a new copy for each *expression* that can be "written" to the variable. Whenever an input-variable is bound by the response to a read_X-move, we syntactically replace any occurrences of that variable by the formula "stored" in the local variable in that copy of the automaton. Again, $\mathsf{write}(E)_X$-moves transition between copies. After performing this construction we hide all X-moves. We illustrate this through an example.

Example 7.6. Consider the SEA $\mathcal{A}$ in Figure 7.7 which recognises the denotation of the program

$$
\begin{array}{l}
x : \mathsf{exp}, y : \mathsf{var} \vdash \\
y := x; \\
\textbf{if}\ !y > 5\ \textbf{then}\ y := 0\ \textbf{else}\ y := y \times 2; \\
!y.
\end{array}
$$

If we wrap this program in a $\textbf{new}\, y\, \textbf{in}$-block we will need to create a new copy of

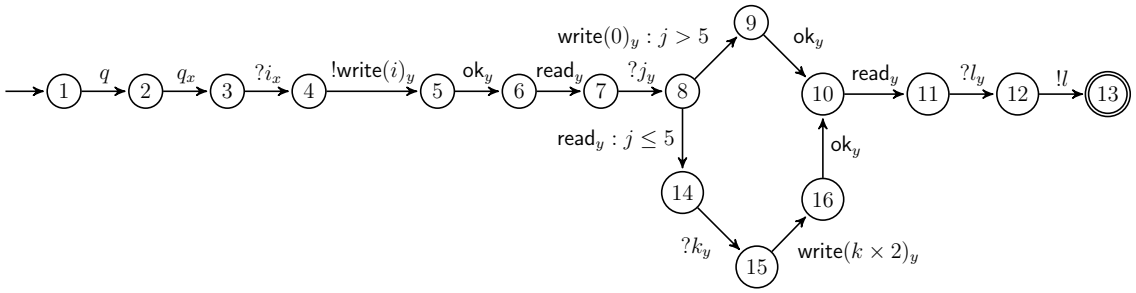


Figure 7.7: Automaton $\mathcal{A}$

$\mathcal{A}$ for each expression which can be "written" to y. Initially, y has the value 0 so we create a corresponding copy of $\mathcal{A}$. We note that the input variables j, k and l are bound by responses to read_ys. Hence, we replace all occurrences of them in this copy of $\mathcal{A}$ with 0. The resulting automaton is shown in Figure 7.8.

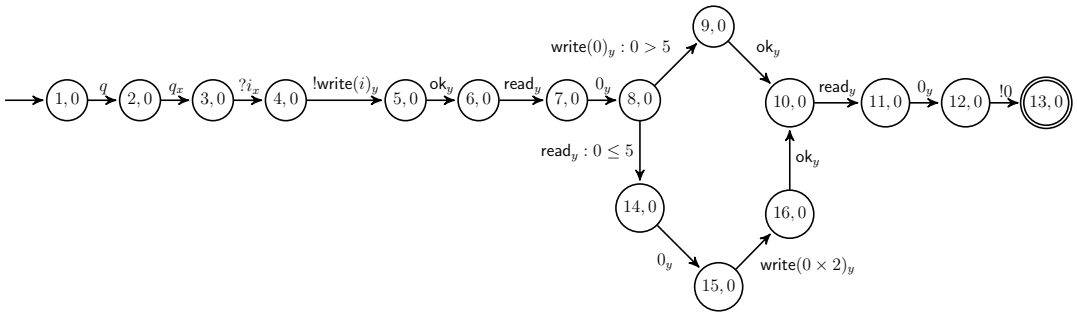


Figure 7.8: The initial copy of $\mathcal{A}$ corresponding to y being 0

We note that this SEA has a (reachable) write$(i)_y$-transition and so we will need to create a fresh copy of $\mathcal{A}$ corresponding to y containing the expression i[1]. In a similar manner as before, all copies of j, k and l in this new copy are replaced by i. Further, the write$(i)_y$-transition is redirected to the appropriate state in the i-valued copy of $\mathcal{A}$. Similarly, the write$(0)_y$-transition in the new copy is redirected back to the 0-valued version of $\mathcal{A}$. The automaton after performing these changes is shown in Figure 7.9.

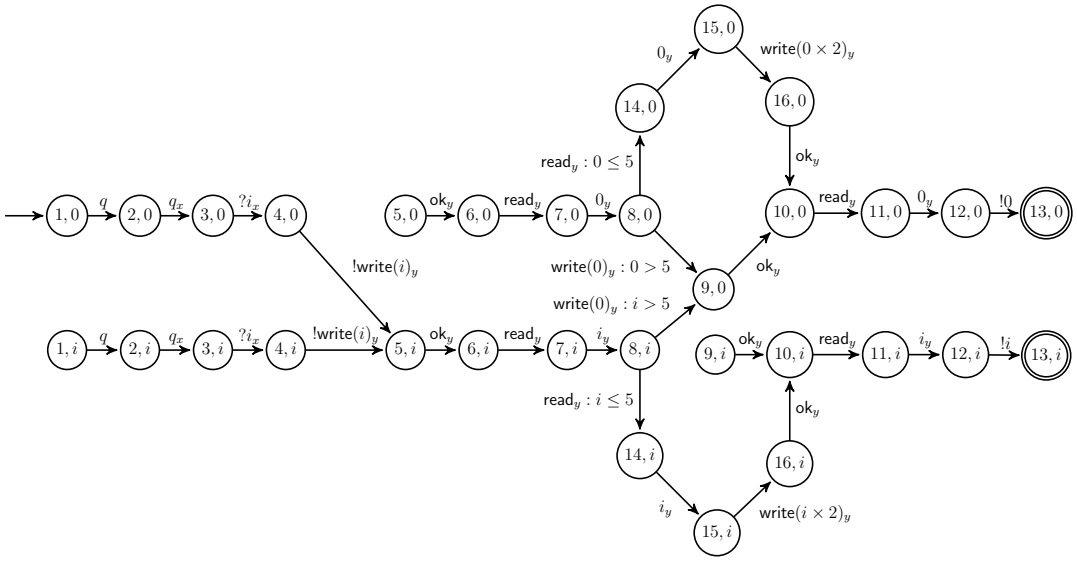


Figure 7.9: After creating copies of $\mathcal{A}$ for 0 and i

We can see that this automaton has a (reachable) write$(i \times 2)_y$-transition. So, we again create a new copy of $\mathcal{A}$, replacing j, k and l by $i \times 2$ and redirecting

[1]We also have write$(0)_y$- and write$(0 \times 2)_y$-transitions. However, these are both equivalent to storing 0 in y for which we already have a copy and furthermore we will see that neither of these transitions will be reachable from the initial state.

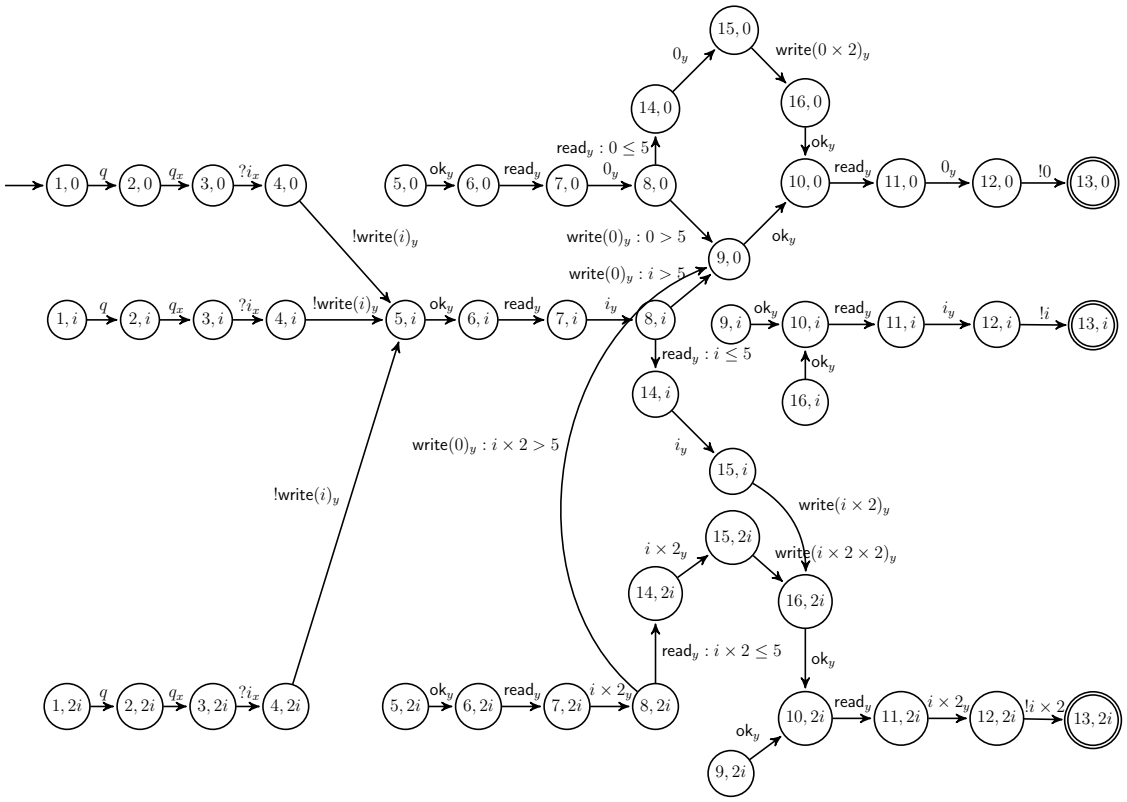


Figure 7.10: After creating copies of $\mathcal{A}$ for 0, i and $2 \times i$

the $\mathsf{write}(i \times 2)_y$-transition into the new copy and the new $\mathsf{write}(0)_y$-transition back into the 0-valued copy, giving the automaton in Figure 7.10. This SEA contains a $\mathsf{write}(i \times 2 \times 2)_y$-transition. However, it is not reachable from the initial state, so we can safely ignore it and we do not need any further copies of $\mathcal{A}$. Now this unrolling has been completed, we just need to hide all the y-transitions. The SEA in Figure 7.11 shows the result of replacing all the y-transitions with ϵ-transitions and removing all unreachable states. Compressing out the ϵ-transitions and moving the (conjunction

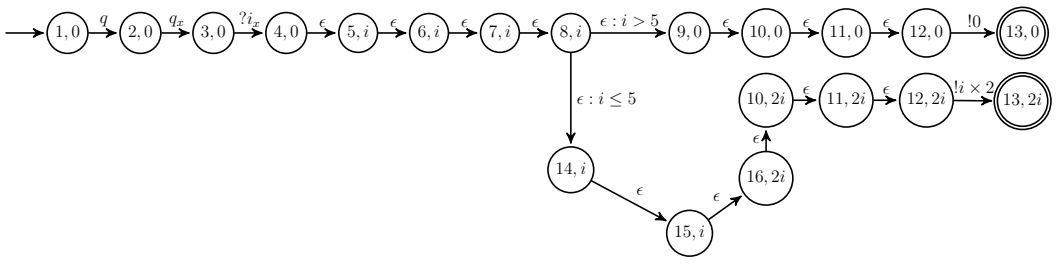


Figure 7.11: SEA resulting from replacing y-moves with ϵ and deleting unreachable states

of) the guards on to the next visible transition gives the final automaton shown in Figure 7.12. This has the form we would have hoped for given the original program.

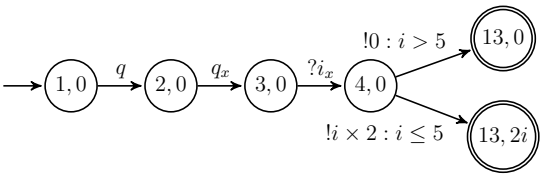


Figure 7.12: Final SEA, created by compressing ϵ-transitions

One subtlety with this construction is that we need to avoid variable capture. The value bound to an input-variable can change if a run contains multiple input-transitions on the same variable. We do not want these changes to affect the value stored in a local variable.

Example 7.7. To illustrate the possibility of variable capture, consider the SEA in Figure 7.13. This SEA recognises the strategy denotation of the term

$$x : \mathsf{exp}, y : \mathsf{var} \vdash y := x + 1; \mathbf{if}\, x > 5\, \mathbf{then}\, !y\, \mathbf{else}\, 0.$$

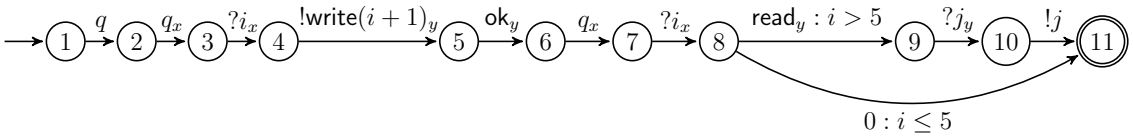


Figure 7.13: SEA with multiple i-input transitions

Note that the automaton has two i-input transitions, either side of a $\mathsf{write}(i+1)_y$-transition. If we naïvely apply our construction for a **new** y **in**-block as before, we create copies of the SEA corresponding to storing 0 (initially) and i in y. The result is shown in Figure 7.14. Unfortunately, this is incorrect. The transition from state $(8, i+1)$ when $i > 5$ is supposed to output the value stored in y. In this state, the expression $i + 1$ was the last expression "written" to y and so the automaton outputs $i+1$. However, the value of i when this was written to y was from the input-transition in state $(3, 0)$. This value has since been overwritten by the transition from state $(7, i+1)$. Hence, the automaton incorrectly accepts the word $q \cdot q_x \cdot 1_x \cdot q_x \cdot 6_x \cdot 7$ but rejects the word $q \cdot q_x \cdot 1_x \cdot q_x \cdot 6_x \cdot 2$ which should be accepted.

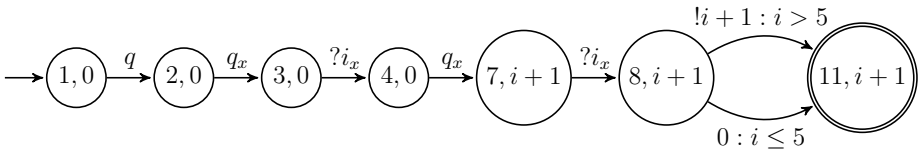


Figure 7.14: Variable capture results from naïvely applying the local variable block construction

In the example above, the variable capture could easily have been avoided by insisting that in the original automaton each input transition (and in particular the transitions out of states 3 and 7) was on a different variable. However, in automata with cycles this does not solve the problem, as a single input-transition can be taken multiple times in a single run. To get around this, we use the same construction as before, but every time we create a new copy of the SEA due to a new expression being written to a variable, we α-convert the input variables in the new copy so that they are distinct from the variable used by existing copies. This solves the problem and results in an automaton which recognises the correct strategy.

In order to produce an automaton with a finite number of states, this method requires there to be only a finite number of distinct expressions that can be written to a variable. For programs without loops this will clearly be the case, as in the previous examples. This will sometimes be the case in the presence of loops as well, as long as we can detect equivalence or unsatisfiability of formulas.

Example 7.8. Consider the program below. We take two inputs, a and b, and store their values in local variables X and Y. Then as long as the value of X is less than or equal to the value of Y we keep switching them. Finally we return the value of X.

```
1  a : exp, b : exp ⊢
2    new X,Y,Z in
3    X := a;
4    Y := b;
5    while !X ≤ !Y do
6    {
7      Z := !X;
8      X := !Y;
9      Y := !Z;
10   };
11   !X
```

Despite the fact that the loop can continue forever (if $a = b$), there will clearly only be a finite number of different formulas that can be stored in each variable, namely

0 (initially), a and b. While this example is rather contrived, there are many other examples, such as sorting a list, which consist of receiving input and then permuting or processing that data in a limited way for which the number of different formulas required will also be finite.

Example 7.9. Recall, in Example 7.6, although we created a copy of the automaton $\mathcal{A}$ for each expression which could be written to y, we ignored the $\mathsf{write}(i \times 2 \times 2)_y$-transition because it was clear that the transition could never be taken as it was unreachable in the underlying control states graph.This is a fairly naïve approach. By utilising the formulas we may be able to detect many more unreachable transitions. Consider the program:

```
1  a : exp ⊢
2    new X,Y,Z in
3    X := a;
4    if !X < 10 then
5    {
6      Z := 0;
7      Y := 0;
8      while !Z ≤ !X do
9      {
10       Z := !Z + 1;
11       Y := !Y + a;
12     }
13   };
14   !Y
```

If we can detect that there is no valid path that loops more than ten times, this will allow us to restrict the number of formulas that Y can take to a finite number. Searching for a valid run for each transition as we construct the model could prove infeasible. However, if we use on-the-fly model checking then we can perform these checks as we are exploring the model, at which time we will have more information on which paths are valid.

Our previous two examples show that even in the presence of loops, our **new** X **in** construction can still produce a finite SEA. Unfortunately, we can construct examples where there is not a finite limit on the number of distinct formulas which can be written to a variable and so our construction fails to produce a finite SEA.

Example 7.10.

```
1  a : exp, b : exp ⊢
2    new X in
3    while b do
4      X := !X + a;
```

Here the number of times we loop is unbounded, determined by the input variable b. Every time through the loop we read an input through a and add it to X. The values in X will have the form $\sum_{i=0}^{k} a_i$ where a_i is the ith value read from a and $k \in \mathbb{N}$. This gives an infinite number of distinct expressions describing the values X can have at different points in the execution and so using our construction will lead to an infinite state space. Note that this occurs even though the SEA for the program before the **new** X **in**-construction is applied is finite and contains only a finite number of expressions; every time we unroll a new copy of the automaton corresponding to a different expression being written to X, it creates new write-transitions on new expressions.

Despite this problem, we hope that for many interesting examples we can use this method to construct (finite) automata. As long as we can detect that only a finite number of different expressions are "written" to a variable in the reachable part of the SEA, the construction will terminate.

7.3 Model Checking SEA

Having translated IA terms into a representation as SEA, the next question is how to equivalence check them. Since SEA are expressively equivalent to finite automata, their equivalence problem is clearly decidable. We consider how to complement and intersect SEA without resorting to converting to finite automata. Then we show how to decide their emptiness problem. This allows us to perform equivalence checking.

Complementation An SEA is *deterministic* if for each word w it has exactly one run on w. Deterministic SEA can be complemented by complementing their set of final states. The automata resulting from our translation will be deterministic by construction.

Intersection We say an automaton $\mathcal{A}$ is *bipartite* if every path in $\mathcal{A}$ strictly alternates between input and output transitions, starting with an input. Since all our plays strictly alternate between O-moves (inputs) and P-moves (outputs), automata resulting from our construction will be bipartite. Given two bipartite automata $\mathcal{A}_1 = \langle\, Q_1, s_1, F_1, \delta_1 \,\rangle$ and $\mathcal{A}_2 = \langle\, Q_2, s_2, F_2, \delta_2 \,\rangle$, we can construct their intersection via a product construction. We assume that the (finite) set of variables used

in the two automata are distinct (this can always be achieved via α-conversion). The resulting automaton will be $\mathcal{A}_{\mathcal{A}_1 \cap \mathcal{A}_2} = \langle\, Q_{\mathcal{A}_1 \cap \mathcal{A}_2}, s_{\mathcal{A}_1 \cap \mathcal{A}_2}, F_{\mathcal{A}_1 \cap \mathcal{A}_2}, \delta_{\mathcal{A}_1 \cap \mathcal{A}_2} \,\rangle$ where

- $Q_{\mathcal{A}_1 \cap \mathcal{A}_2} = Q_1 \times Q_2$.
- $s_{\mathcal{A}_1 \cap \mathcal{A}_2} = (s_1, s_2)$.
- $F_{\mathcal{A}_1 \cap \mathcal{A}_2} = F_1 \times F_2$.
- If $q_1 \overset{?X_1 : \psi_1}{\longrightarrow} q_1' \in \delta_1$ and $q_2 \overset{?X_2 : \psi_2}{\longrightarrow} q_2' \in \delta_2$, then $(q_1, q_2) \overset{?(X_1 \cup X_2) : (\psi_1 \wedge \psi_2)}{\longrightarrow} (q_1', q_2') \in \delta_{\mathcal{A}_1 \cap \mathcal{A}_2}$.
- If $q_1 \overset{!\phi_1 : \psi_1}{\longrightarrow} q_1' \in \delta_1$ and $q_2 \overset{!\phi_2 : \psi_2}{\longrightarrow} q_2' \in \delta_2$, then $(q_1, q_2) \overset{!\phi_1 : ((\phi_1 = \phi_2) \wedge \psi_1 \wedge \psi_2)}{\longrightarrow} (q_1', q_2') \in \delta_{\mathcal{A}_1 \cap \mathcal{A}_2}$.

Then $\mathcal{A}_{\mathcal{A}_1 \cap \mathcal{A}_2}$ has an accepting run on w if and only if both $\mathcal{A}_1$ and $\mathcal{A}_2$ have accepting runs on w.

Emptiness With a finite automaton we can check for emptiness by using a breadth-first search. We maintain a set of visited states and a queue of states we need to process. At each step we take the front element of the queue, say q, and for each state q' reachable by a transition from q, if q' is not in the set of visited states then we add it to that set and also to the end of the queue. If we ever reach a final state we return "non-empty". If we get to the point of having an empty queue without encountering a final state then we return "empty".

With SEA the situation is a little more complex because we may wish to revisit an already visited state if the valuation function will be different from the previous visit. To overcome this, instead of maintaining a set of visited states, we associate with each state q a formula ϕ_q such that if for a valuation V, $V \models \phi_q$ then we know that there is some run π on w which ends in q and $val_{\pi,w} = V$. To simplify things, we add a special $\perp$ value which is not in Σ. If $V(x) = \perp$ then it should be taken to mean $val_{\pi,w}(x)$ is not defined for a particular run that led to this state (i.e. that run does not pass through an x-input transition). Initially $\phi_q = \mathit{false}$ for all non-initial q. For the initial state, $\phi_{q_0} = \bigwedge_x (x = \perp)$, where the conjunction is for all variables occurring in the automaton.

When performing a breadth-first search of a finite automaton, when we visit a state we add all unvisited neighbours to the queue of states to explore. Instead,

when searching an SEA we will construct for each outgoing transition a new formula ψ_q defining what the new ϕ_q should be if we take this transition into account. We will then need to see whether this adds any new valuations. To do this we define a partial order on formulas: $\phi \leq \psi$ if and only if $\phi \Rightarrow \psi$ is valid. Then $\phi < \psi$ if and only if $\phi \leq \psi \wedge \neg(\psi \leq \phi)$. Note that $\phi < \psi$ if and only if $\{V \mid V \models \phi\} \subset \{V \mid V \models \psi\}$.

Now we can give our emptiness algorithm which is shown in Figure 7.15. When we encounter an input transition, the new formula uses existential quantification to ensure that the path up to the previous state was valid but that in the new state the input variables can have a different value (not $\perp$) but must satisfy the guard. For output transitions the valuation does not change and so we just check that the previous state can be reached and the guard is valid. Each time we assign a new

```
1   while queue non-empty
2     q := dequeue(queue)
3     if q ∈ F return "non-empty"
4     for each outgoing transition e = q --T:ψ--> q'
5       if e = q --?X:ψ--> q'
6         Ψ := φ_q' ∨ ((∃X.φ_q) ∧ ψ ∧ (⋀_{x∈X}(x ≠ ⊥))
7       else if e = q --!E:ψ--> q'
8         Ψ := φ_q' ∨ (φ_q ∧ ψ)
9
10      if Ψ > φ_q' then
11        φ_q' := Ψ
12        enqueue(q', queue)
13  return "empty"
```

Figure 7.15: Reachability algorithm for SEA

value to a ϕ_q, the new formula has more satisfying valuations than the old one. Since there are a finite number of valuations (there are a finite number of variables with a finite number of values) the algorithm must terminate.

7.4 Implementation

We have a working F# implementation of the ideas presented in this chapter. We follow an on-the-fly approach so that we can detect when a path is no longer worth exploring and so often limit models to be finite. Since it is often the case that a transition (whether input or output) will be constrained to only take a single value,

we make this a special case of transition. This simplifies the guards required which in turn simplifies the formulas we have to ask the SMT solver about. Additionally, we have to allow ϵ-transitions. While we might like to compress out ϵ-transitions by pushing the guard on to the next visible transition (as we did in Example 7.6) this does not always work. The problem is that we may have an infinite chain of ϵ-transitions with unsatisfiable guards. Clearly attempting to move the guards onto the next visible transition will not work. However, making the ϵ-transitions "visible" to the exploration algorithm allows us to use the SMT solver to determine that the guards are unsatisfiable and so we know that we should ignore this path. Unfortunately, the presence of the ϵ-transitions can drastically increase the state-space (since paths can be much longer). This could be particularly bad in the product automaton used for the symmetric difference construction in the equivalence check. This is because all the ϵ-transitions from the two automata do not require any synchronisation and so the state space multiplies. However, since it does not make any difference in which order the ϵ-transitions in the two automata are explored we can apply some symmetry reduction and enforce an ordering to stop this occurring.

For the SMT solver we use Z3, [34]. This has the advantages of being an efficient theorem prover which supports our required language and having a .NET interface. Unfortunately, we had difficulty accessing Z3's .NET API under mono (which is used to run .NET programs on Linux) and so the performance data presented in this chapter refers to execution on a different machine than that used for the previous chapter. In the following sections, our implementation is tested and compared to HOMER running on a laptop with a 2.53GHz Intel Core 2 Duo processor and 4GB RAM under Windows Vista.

7.4.1 Identities

We motivated SEA by noting that the explicit state representation fails to take advantage of the symmetries of the game-semantic model. This is particularly evident when considering the identity functions. In the game-semantic model these are represented as copycat strategies and have a very simple description. However, as we have seen, if the base types are very large the explicit state representation can be excessively big. Hence, we would hope this is an example on which the SEA approach would be successful.

Consider the two identity functions, $x : \mathsf{exp} \vdash x$ and $x : \mathsf{exp} \vdash \mathbf{new}\, X\, \mathbf{in}\, X := x; !X$. The running times for our SEA implementation are shown in Figure 7.1 for various sizes of the base type exp. The value n refers to the size of exp (i.e. if $n = 32$ this

refers to 32-bit integers). We also list the number of states found, the number of calls to the SMT solver and the total amount of time spent waiting for the SMT solver. For comparison we also give the running time of HOMER and the final and maximum number of states that HOMER uses. It can be seen that, as we expected, using SEA drastically speeds up equivalence checking this example. The running time is hardly affected by the size of exp. By contrast, as the size of exp increases, the number of states HOMER has to consider explodes and consequently the running times are much slower.

n	SEA	States	Z3 Calls	Z3 time	HOMER	Final Size	Max Size
2	.6s	7	6	.001s	0.8s	11	32
5	.7s	7	6	.001s	1s	67	1,200
10	.6s	7	6	.001s	5min	2,100	1,100,000
32	.7s	7	6	.001s	TIMEOUT	TIMEOUT	TIMEOUT

Table 7.1: Running times on the identity example

7.4.2 Sorting

We now return to the example of sorting lists. Previously when we used this example, we only considered varying the length of the list. Since we hope SEA will help deal with large alphabets, we now consider varying both the length of the list and the size of the integers stored in the list. The results are shown in Table 7.2. Again n refers to the size of the integers. As can be seen, the SEA approach allows the size of the data stored in the list to be increased well beyond the size HOMER can deal with. However, on some of the cases with 2-bit integers, the symbolic approach does not pay off and HOMER is faster. We can also see that the number of calls made to the SMT solver grows rapidly with the size of the list. As the size of the integers increase, these calls come to dominate the running time.

7.4.3 Summing

As another example involving a loop, we consider the programs in Figure 7.16. The first program reads its input into Z and checks whether it is less than N. If it is, the program then calculates the sum of all integers less than or equal to Z using a loop and outputs the result. The second program instead immediately outputs $((!Z + 1) * !Z) >> 1$. By summing the arithmetic series we can see that these two programs should be equivalent. The game-semantic model will need to calculate the output for every possible input. Hence, the value of N plays a major role in

n	SEA	States	Z3 Calls	Z3 time	HOMER	Final Size	Max Size
Lists of Length 2							
2	.7s	34	37	.002s	1.4s	71	270
5	.7s	34	37	.002s	54s	3,200	38,000
10	.8s	34	37	.004s	TIMEOUT	TIMEOUT	TIMEOUT
32	.8s	34	37	.02s	TIMEOUT	TIMEOUT	TIMEOUT
Lists of Length 5							
2	8s	5,800	8,200	.03s	5.7s	1,400	17,000
5	9s	5,800	8,200	1s	TIMEOUT	TIMEOUT	TIMEOUT
10	11s	5,800	8,200	3s	TIMEOUT	TIMEOUT	TIMEOUT
32	22s	5,800	8,200	13.5s	TIMEOUT	TIMEOUT	TIMEOUT
Lists of Length 7							
2	9min	290,000	420,000	12s	90s	13,000	270,000
5	15min	400,000	570,000	102s	TIMEOUT	TIMEOUT	TIMEOUT
10	18min	400,000	570,000	5min	TIMEOUT	TIMEOUT	TIMEOUT
32	38min	400,000	570,000	24min	TIMEOUT	TIMEOUT	TIMEOUT

Table 7.2: Running times for comparing bubble-sort and insertion-sort

the complexity of the problem. The running times for this example are shown in Table 7.3. These were performed under the assumption that all integers were 32-bit. By contrast, HOMER could not handle this example even for $N = 10$ and with all integers as small as possible while avoiding overflow.

N	Time	States	Z3 Calls	Z3 time
10	1s	80	160	0.1s
100	3s	620	1,300	1s
1,000	51s	6,000	13,000	19s
5,000	950s	30,000	65,000	300s
10,000	TIMEOUT	TIMEOUT	TIMEOUT	TIMEOUT

Table 7.3: Running times for comparing summing programs

This example shows that as well as comparing programs with loops and very different control flow, we can also handle programs which perform arithmetic. However, on some examples the SMT solver struggles to cope with the arithmetic. For example, consider the very simple program $x : \mathsf{exp} \vdash 207 * x$. Comparing this to itself (assuming 32-bit integers) requires checking only 7 states. However, the check takes 3 minutes, almost all of which time is taken up waiting for the SMT solver to perform 6 satisfiability tests. In particular checking the seemingly innocuous formula $x = y \wedge \neg(x \times 207 = y \times 207)$ takes Z3 40 seconds to determine unsatisfiable. If the multiplication is part of a larger program, the formulas can be much more complicated

```
1   input : exp ⊢
2   new X in new Y in new Z in
3   {
4     Z := input;
5     if !Z > N then 0 else {
6     while !Y < !Z do
7     {
8       Y := !Y +1;
9       X := !X + !Y
10    };
11    !X
12    }
13  }
```

```
1   input : exp ⊢
2   new X in new Z in
3   {
4     Z := input;
5     if !Z > N then 0 else {
6     X := ((!Z + 1) * !Z) >> 1;
7     !X
8     }
9   }
```

Figure 7.16: Two programs for calculating a sum

and the SMT solver can be overwhelmed.

7.4.4 Wavelet

As a final example we consider the Discrete Wavelet Transform example used in [65]. The program is shown in Figure 7.17. It uses several loops to populate, manipulate and then output the contents of an array of 100 8-bit integers. We consider two variants. One in which loop fusion has been used to reduce the number of while loops and another which has additional optimisations using scalar replacement. We compare these both when run on a fixed input and when run on arbitrary input. The results are shown in Table 7.4.

Prog. 1/Input	Prog. 2/Input	Time	States	Z3 Calls	Z3 time
Original/Fixed	Fusion/Fixed	10s	200	200	0.002s
Original/Fixed	Scalar Replacement/Fixed	11s	200	200	0.002s
Fusion/Fixed	Scalar Replacement/Fixed	11s	200	200	0.002s
Original/Any	Fusion/Any	533s	600	600	520s
Original/Any	Scalar Replacement/Any	525s	600	600	512s
Fusion/Any	Scalar Replacement/Any	530s	600	600	514s

Table 7.4: Running times for comparing the wavelet examples

These examples contain a large state space (the loop fusion optimisation involves four arrays of 100 integers) and differing control flows (the original involves five processing loops, whereas the other two versions only have two). While we cannot compete with the results listed in [65] (though it is not clear which variant of the problem is considered there, nor how much manual preprocessing their tool requires), the fact that we can handle these examples is very encouraging as they are far beyond the size HOMER can cope with. It is interesting (if not necessarily surprising) to note that the extra running time caused by allowing arbitrary input rather than considering a fixed input is almost entirely spent waiting for the SMT solver. This might suggest that any future work targeting improved performance on this sort of example should focus on heuristics to optimise the queries sent to Z3.

7.5 Summary

In this chapter we have introduced the idea of Symbolically Executed Automata. These are automata in which the transitions are labelled with formulas and guards instead of concrete letters. They are intended as a compact representation which

```
1  io : var ⊢
2  new i in new a[100] in new b[100] in
3  { i := 0;
4    while !i < 100 do
5    {   a[!i] := !io; i := !i + 1 };
6    i := 0;
7    while !i < 50 do
8    { if(!i=49) then
9        a[2*!i+1] := !a[2*!i+1] − (!a[2*!i] + !a[2*!i])>>1
10     else
11       a[2*!i+1] := !a[2*!i+1] − (!a[2*!i] + !a[2*!i+2])>>1;
12     i := !i + 1
13   };
14   i := 0;
15   while !i < 50 do
16   { if(!i=0) then
17       a[2*!i] := !a[2*!i] + (!a[1] + !a[2*!i+1] + 2)>>2
18     else
19       a[2*!i] := !a[2*!i] + (!a[2*!i−1] + !a[2*!i+1] +2)>>2;
20     i := !i + 1
21   };
22   i := 0;
23   while !i < 50 do
24   { b[!i]:=!a[2*!i];
25     i := !i + 1
26   };
27   i := 0;
28   while !i < 50 do
29   { b[50+!i]:= !a[2*!i+1];
30     i := !i + 1
31   };
32   i := 0;
33   while !i < 100 do
34   { a[!i]:=!b[!i];
35     i := !i + 1
36   };
37   i := 0;
38   while !i < 100 do
39   { io := !a[!i];i := !i + 1
40   }
41 }
```

Figure 7.17: Wavelet original

fits naturally with the game-semantic model, taking advantage of its symmetries and alleviating the problem of requiring large alphabets. We have discussed how to translate IA programs into SEA and shown that often (but not always) this can be done automatically in a way which gives small (finite) automata. These automata can then be checked for equivalence which requires calling an SMT solver. We have implemented these ideas and tested the resulting tool on a number of examples. The symbolic approach allows us to handle much larger data types than are possible using an explicit state representation, including 32-bit integers and arrays containing hundreds of elements. However, on examples with small data types but complex control flow, the explicit state approach wins out.

Our implementation accepts terms of first-order IA. It should be straightforward to extend this to second-order IA as the game semantics is still regular. Extending further to third-order would present more of a challenge as pushdown automata would be needed. However, it may still be possible to augment our notion of SEA with a stack and perhaps consider symbolically executed visibly pushdown automata. Another possible way to extend the range of examples we can handle would be to modify the compilation algorithm. As noted, on some examples the compilation will not produce a finite SEA, so the algorithm is not a decision procedure. It may be possible to find some heuristic to compromise between storing expressions stored in a local variable and storing concrete values. In some cases this might allow us to produce a finite representation which is still relatively compact.

The other direction further work could take would be on improving the performance. On many examples, the SMT solver proves to be a bottle neck. However, in some cases the problem is that we perform a very large number of (often fairly simple) calls to Z3 while in other cases only a few calls are made but they are particularly difficult for Z3 to handle. It would be interesting to consider whether it were possible to find a balance between performing many calls and performing complex calls. Further, it may be possible to find a way to phrase the queries in a way that the SMT solver finds easier to deal with.

Related Work Veanes et al. have proposed a similar framework to ours [107, 106, 105]. They also attempt to represent finite automata over large alphabets by labelling transitions with symbolic expressions rather than concrete values. Transitions in their *Symbolic Finite Automata* (SFA) are labelled with a formula containing a single free variable. A transition can be followed on a given input letter if substituting the letter for the free variable results in a true formula. This allows them to perform constraint

solving and recognise, for example, strings which have the form of an email address in a very compact and efficient manner. SFA, though, do not contain input-transitions in the sense of SEA. A transition cannot refer to a value that occurred in a previous transition of a run. This makes SFA inappropriate for recognising our motivating example of the identity strategy, as we require consecutive transitions to be on the same value (in different components). Our SEA (which were developed independently and, dating from 2009, predate their publications) were designed with capturing game-semantic strategies in mind and the ability to refer to the value of previous transitions is crucial to this.

Chapter 8

Conclusion

In this thesis we have explored the use of game semantics for the equivalence checking of higher-order programs. Game semantics provides highly accurate models of higher-order programming languages by viewing the execution of a program as the playing of a two player game between the program and its environment. By taking advantage of the concrete nature of the semantics and representing the strategy denotation of terms using automata, we can decide observational equivalence of fragments of languages. Thanks to the fully abstract nature of the game-semantic models, this methodology is both sound and complete. Further, as we can consider open terms, it is also compositional. This approach is fully automatic and does not require any annotation on the part of the user.

8.1 RML

We started out by considering the call-by-value higher-order language with ground type references RML [5]. We identified the O-strict fragment of RML. That is, the largest fragment in which (in RML's fully abstract game-semantic model) the location of justification pointers from O-moves is always uniquely determined. We showed that this fragment consists of terms of short-type, by which we mean order at most two and arity at most one. Further, terms may contain free identifiers as long as all their argument types are also short. This means large higher-order type sequents are included (order-three free identifiers of arbitrary arity are allowed so long as their arguments are short). The fragment includes many examples which are known to be difficult to reason about. We showed that the strategy denotations of terms can be precisely represented as languages of finite words over a finite alphabet. To do this we needed to encode the location of P-pointers which we did using a single-pointer representation. Each word encoded the location of at most one P-pointer, but every

ambiguous pointer was encoded by some word. Hence, when the language as a whole was considered, the location of every P-pointer could be recovered. Further, we showed that the languages denoting O-strict terms could be recognised by VPA. We constructed the VPA inductively over the structure of the normal forms of RML. This is sufficient to show that observational equivalence of O-strict RML is decidable. Our constructions could be performed in ExpTime. We went on to show that the problem is ExpTime-complete by reduction of the equivalence problem for nondeterministic automata on binary trees. These results were presented in [47].

We then considered extensions to the O-strict fragment. Our initial decidability result was for RML with loops but without any other form of recursion. We showed, in the style of [76], that first-order recursive functions of arity one could be added while preserving decidability. However, we needed to use DPDA rather than VPA. This increase in power from VPA to DPDA really was necessary as we were able to show that the problem is as hard as the DPDA equivalence problem.

The next extension we considered was to allow types that were no longer O-strict. Encoding O-pointers is a lot more challenging than encoding P-pointers. This is because O-pointers are controlled by the environment and as we must consider all possible environments we have to allow all possible locations for O-pointers. In contrast, P-pointers are controlled by the term so the strategy for a given term will contain plays with only one pointer location. Due to this, our previous single-pointer representation does not work for O-pointers. Instead, to encode plays we had to use infinite alphabets. We showed how strategy denotations of terms of order one and arity two (with free variables whose arguments were at most order one and arity one) could be encoded as data languages [97]. Furthermore, we showed how to recognise such languages using deterministic CMA [21] and how such automata could be checked for equivalence.

Completing our analysis of RML, we finally considered when observational equivalence becomes undecidable. We showed that order-three types or types of order two in which an argument other than the last was first-order can be used to show undecidability. Furthermore, free identifiers which take an argument of such a type are also enough to make the problem undecidable. The proofs rely on encoding the computation of a Q-machine [72] (equivalent to a finite automaton equipped with a queue) using O-pointers to simulate the Q-store's FETCH action. Additionally, second-order recursive functions can also be used to simulate a queue and so lead to undecidability.

A summary of the known decidable and undecidable fragments was presented in Table 4.2. We originally hoped to gain a complete picture of the decidable fragments of RML as has been achieved for IA. We have made large amounts of progress towards this aim. Our decidable fragments greatly extend those of [40, 73], allowing precise representations of higher-order functions. Further, our undecidability results narrow the gap from the other side. Unfortunately, though, we have not quite managed to complete the picture. There are still a small number of type sequents which we have not yet been able to show either decidable or undecidable. An obvious goal of future work is to resolve this. These unknown cases are all non-O-strict. They do not seem to have enough expressivity to encode a Q-machine in the same manner as our existing undecidability proofs. However, it is also not clear what class of automata would be suitable for recognising their semantics. Some of the unresolved cases seem to require VPA (or DPDA in the case of recursion) over infinite alphabets. Others appear to need some form of nesting of data values. We are not currently able to ascertain whether automata exist which can recognise such languages yet have a decidable equivalence problem.

We would also like to know the exact complexity of deciding observational equivalence for $\mathsf{RML}_{\mathrm{CMA}}$. It is not yet clear whether our use of CMA results in an optimal complexity bound or not.

8.2 Non-Local Control Flow

In Chapter 5 we moved on to looking at languages with non-local control flow. Game semantically, allowing non-local jumps of control corresponds to dropping the bracketing condition. We first looked at IA augmented with **catch**. In the game-semantic model of IA, answers can never be used to justify another move. Because of this, when bracketing is removed the visibility condition still implies a weak bracketing condition [60]. This means that although an answer does not have to answer the most recently asked question, it cannot answer a question if another earlier asked question has already been answered. Essentially, non-local jumps in control can pop the call stack, but cannot "pop the stack upwards". This allowed us to encode the location of ambiguous justification pointers which violate bracketing by making the violations explicit and to capture strategies using VPA which pop moves off the stack when they are closed. In doing this we showed that decidability of third-order IA is preserved when **catch** is added to the language. Further, by using a product construction to

recognise an alternative ordering on strategies, we showed that this result holds for the language both with and without a bad-variable constructor [75].

For future work we would like to find decidable fragments of RML with non-local control flow. Unfortunately, this appears more challenging than in the call-by-name case. Since answers can justify questions in the call-by-value model, the visibility condition is no longer sufficient to imply weak bracketing. This means that control flow jumps no longer just correspond to popping the call stack and so we cannot encode bracketing violations in the same manner. It does not appear that pushdown automata are suitable for capturing strategies of this language, as there is little about the semantics which resembles a stack-like behaviour. It is possible that some form of automata with infinite alphabets may be able to capture a class of terms, in the same manner as for well-bracketed RML. Any further progress on capturing non-O-strict well-bracketed strategies may also feed directly into this.

It would also be interesting to investigate decidability results for other language extensions. Recursive types and general references are both standard features of ML-like languages and can be captured using game semantics [66, 2]. Unrestricted, their addition to our languages would immediately lead to undecidability. However, it may be possible to find some natural conditions on their use to recover decidability. Similarly, rather than completely dropping bracketing from the call-by-value model, there may be a weaker form of non-local control flow which could be added to RML. Weak bracketing is not preserved by composition in the call-by-value model, but there may be a different condition (and corresponding control construct) which is preserved and can be used to find a decidable fragment.

8.3 Implementation

We also presented our tools HOMER [49] and HECTOR [48] which are equivalence checkers for the third-order fragment of IA and the O-strict fragment of RML respectively. These both construct VPA precisely representing the game-semantic denotations of terms. The constructed VPA are language equivalent if and only if the terms they represent are observationally equivalent. Both tools rely on an explicit state representation but HECTOR uses on-the-fly model checking. This avoids constructing unreachable parts of the VPA and also allows early termination if a counterexample is found. Both tools are easily able to handle many challenging examples. Compared to the game semantics based model checker MAGE [12], they are outperformed (unsurprising given that MAGE checks simpler properties and uses abstraction techniques)

but are encouragingly competitive. However, due to their explicit state representations, on larger examples they run into the state space explosion problem. To help deal with large data types we proposed SEA, adopting ideas from symbolic execution. We showed how to represent the game semantics of terms as a form of automata with transitions labelled with formulas and guards. This allowed a more compact representation, particularly reducing the size of our automata's alphabets. For simplicity we only considered first-order programs but the idea should work for higher-order programs too. Our implementation relied on the SMT solver Z3 [34] to check satisfiability of formulas. Using SEA allowed us to handle much larger data types (32-bit integers). However, the compilation to SEA was not guaranteed to produce a finite automaton and so the method was incomplete. It also struggled with the state space explosion problem on larger examples and gave little benefit when only small data types were used.

One direction of future work would be to try to find ways to optimise the SEA implementation. On some examples, the tool makes an excessive number of calls to the SMT solver or asks about formulas that Z3 finds hard to deal with. It may be possible to find heuristics to cut back on the number of calls or rephrase formulas to speed up the process.

Our current algorithm for constructing SEA does not always produce a finite automaton. On the examples we tested our tool on, this was not a problem. However, it may also be worth exploring compromises between the symbolic and explicit approaches to try to find a compilation algorithm which is both compact and complete. We would also like to extend the input language to include higher-order types and potentially integrate the result into HOMER.

Finally, our tools would also benefit from other model checking techniques. Methods such as data abstraction [12], predicate abstraction [13] and CEGAR [35] have been used in a game-semantic setting, but only for reachability checking. Adapting these for equivalence checking would be non-trivial but might improve performance.

Bibliography

[1] S. Abramsky, D. R. Ghica, A. S. Murawski, and C.-H. L. Ong. Applying game semantics to compositional software modeling and verification. In *TACAS*, 2004.

[2] S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In *LICS*, 1998.

[3] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. In *TACS*, 1994.

[4] S. Abramsky and G. McCusker. Games for recursive types. In *Theory and Formal Methods of Computing*, 1994.

[5] S. Abramsky and G. McCusker. Call-by-value games. In *CSL*, 1997.

[6] S. Abramsky and G. McCusker. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. In *Algol-like languages, Birkhauser*, 1997.

[7] S. Abramsky and G. McCusker. Full abstraction for Idealized Algol with passive expressions. *Theor. Comput. Sci.*, 227(1-2), 1999.

[8] S. Abramsky and G. McCusker. Game semantics. In *Computational Logic: Proceedings of the 1997 Marktoberdorf Summer School*, 1999.

[9] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *POPL*, 2009.

[10] R. Alur, S. Chaudhuri, K. Etessami, and P. Madhusudan. On-the-fly reachability and cycle detection for recursive state machines. In *TACAS*, 2005.

[11] R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, 2004.

[12] A. Bakewell and D. R. Ghica. On-the-fly techniques for game-based software model checking. In *TACAS*, 2008.

[13] A. Bakewell and D. R. Ghica. Compositional predicate abstraction from game semantics. In *TACAS*, 2009.

[14] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, 2001.

[15] T. Ball and S. K. Rajamani. The SLAM toolkit. In *CAV*, 2001.

[16] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *CSF*, 2008.

[17] N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *TLCA*, 2005.

[18] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, 1999.

[19] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *CoRR*, abs/cs/0610081, 2006.

[20] H. Björklund and M. Bojańczyk. Shuffle expressions and words with nested data. In *MFCS*, 2007.

[21] H. Björklund and T. Schwentick. On notions of regularity for data languages. *Theor. Comput. Sci.*, 411(4-5), 2010.

[22] M. Bojańczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 12(4), 2011.

[23] R. E. Bryant. A methodology for hardware verification based on logic simulation. *Journal of the ACM*, 38:299–328, 1991.

[24] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3), 1992.

[25] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond, 1990.

[26] R. Cartwright and M. Felleisen. Observable sequentiality and full abstraction. In *POPL*, 1992.

[27] A. Cataldo. *The Power of Higher-Order Composition Languages in System Design*. PhD thesis, University of California, 2006.

[28] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, 2000.

[29] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2001.

[30] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, 2004.

[31] E. M. Clarke and H. Veith. Counterexamples revisited: Principles, algorithms, applications. In *Verification: Theory and Practice*, 2003.

[32] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[33] D. Currie, X. Feng, M. Fujita, A. J. Hu, M. Kwan, and S. Rajan. Embedded software verification using symbolic execution and uninterpreted functions. *Int. J. Parallel Program.*, 34(1), 2006.

[34] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[35] A. Dimovski, D. R. Ghica, and R. Lazic. Data-abstraction refinement: A game semantic approach. In *SAS*, 2005.

[36] D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *ICFP*, 2010.

[37] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103:235–271, 1992.

[38] M. P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps.* Cambridge University Press, 1996.

[39] T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI*, 1991.

[40] D. R. Ghica. Regular-language semantics for a call-by-value programming language. In *MFPS*, 2001.

[41] D. R. Ghica and A. Bakewell. Clipping: A semantics-directed syntactic approximation. In *LICS*, 2009.

[42] D. R. Ghica and G. McCusker. Reasoning about Idealized Algol using regular languages. In *ICALP*, 2000.

[43] B. Godlin and O. Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Inf.*, 45(6), 2008.

[44] B. Godlin and O. Strichman. Regression verification. In *DAC*, 2009.

[45] C. A. R. Hoare. The verifying compiler, a grand challenge for computing research. In *VMCAI*, 2005.

[46] K. Honda and N. Yoshida. Game theoretic analysis of call-by-value computation. In *ICALP*, 1997.

[47] D. Hopkins, A. S. Murawski, and C.-H L. Ong. A fragment of ML decidable by visibly pushdown automata. In *ICALP*, 2011.

[48] D. Hopkins, A. S. Murawski, and C.-H L. Ong. HECTOR: An equivalence checker for a higher-order fragment of ML. In *CAV*, 2012.

[49] D. Hopkins and C.-H L. Ong. HOMER: A higher-order observational equivalence model checker. In *CAV*, 2009.

[50] A. J. Hu. High-level vs. RTL combinational equivalence: An introduction, 2006.

[51] G. Hutton. Higher-order functions for parsing. *J. Funct. Program.*, 2(3), 1992.

[52] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2), 2000.

[53] R. Jhala, R. Majumdar, and A. Rybalchenko. HMC: Verifying functional programs using abstract interpreters. In *CAV*, 2011.

[54] N. D. Jones and N. Andersen. Flow analysis of lazy higher-order functional programs. *Theor. Comput. Sci.*, 375(1-3), 2007.

[55] N. D. Jones and S. S. Muchnick. The complexity of finite memory programs with recursion. *J. ACM*, 25(2), 1978.

[56] M. Kaminski and N. Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2), 1994.

[57] N. Kobayashi. Model-checking higher-order functions. In *PPDP*, 2009.

[58] N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *POPL*, 2009.

[59] D. Kroening and E. M. Clarke. Checking consistency of C and Verilog using predicate abstraction and induction. In *ICCAD*, 2004.

[60] J. Laird. Full abstraction for functional languages with control. In *LICS*, 1997.

[61] J. Laird. *A Semantic Analysis of Control*. PhD thesis, University of Edinburgh, 1999.

[62] J. Laird. A game semantics of local names and good variables. In *FOSSACS*, 2004.

[63] R. Loader. Finitary PCF is not decidable. *Theor. Comput. Sci.*, 266(1-2), 2001.

[64] H. G. Mairson. A simple proof of a theorem of Statman. *Theor. Comput. Sci.*, 103, 1992.

[65] T. Matsumoto, K. Seto, and M. Fujita. Formal equivalence checking for loop optimization in C programs without unrolling. In *ACST*, 2007.

[66] G. McCusker. Games and full abstraction for FPC. In *LICS*, 1996.

[67] G. McCusker. On the semantics of the bad-variable constructor in Algol-like languages. In *MFPS*, 2003.

[68] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, 1992.

[69] J. Midtgaard. Control-flow analysis of functional programs. Technical report, BRICS, 2007.

[70] M. L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., 1967.

[71] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1), 1991.

[72] A. S. Murawski. On program equivalence in languages with ground-type references. In *LICS*, 2003.

[73] A. S. Murawski. Functions with local state: regularity and undecidability. *Theor. Comput. Sci.*, 338(1-3), 2005.

[74] A. S. Murawski. Games for complexity of second-order call-by-name programs. *Theor. Comput. Sci.*, 343(1-2), 2005.

[75] A. S. Murawski. Bad variables under control. In *CSL*, 2007.

[76] A. S. Murawski, C.-H. L. Ong, and I. Walukiewicz. Idealized Algol with ground recursion, and DPDA equivalence. In *ICALP*, 2005.

[77] A. S. Murawski and N. Tzevelekos. Full abstraction for Reduced ML. In *FOSSACS*, 2009.

[78] A. S. Murawski and N. Tzevelekos. Block structure vs. scope extrusion: Between innocence and omniscience. In *FOSSACS*, 2010.

[79] A. S. Murawski and N. Tzevelekos. Algorithmic nominal game semantics. In *ESOP*, 2011.

[80] A. S. Murawski and N. Tzevelekos. Game semantics for good general references. In *LICS*, 2011.

[81] A. S. Murawski and I. Walukiewicz. Third-order Idealized Algol with iteration is decidable. In *FOSSACS*, 2005.

[82] F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Logic*, 5(3), 2004.

[83] H. Nickau. Hereditarily sequential functionals. In *LFCS*, 1994.

[84] P. W. O'Hearn and J. C. Reynolds. From Algol to polymorphic linear lambda-calculus. *J. ACM*, 47(1), 2000.

[85] C. Okasaki. Even higher-order functions for parsing. *J. Funct. Program.*, 8(2), 1998.

[86] C.-H L. Ong. An approach to deciding the observational equivalence of Algol-like languages. *Annals of Pure and Applied Logic*, 130, 2004.

[87] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS*, 2006.

[88] C.-H. L. Ong and C. A. Stewart. A Curry-Howard foundation for functional computation with control. In *POPL*, 1997.

[89] M. Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In *LPAR*, 1992.

[90] A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. *Higher order operational techniques in semantics*, 1998.

[91] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theor. Comput. Sci.*, 1(2), 1975.

[92] G. D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3), 1977.

[93] E. L. Post. Formal reductions of the general combinatorial decision problem. *AJM*, 65(2), 1943.

[94] J. C. Reynolds. The essence of Algol. In *Algorithmic Languages*, 1981.

[95] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *PLDI*, 2008.

[96] S. Schwoon. *Model-Checking Pushdown Systems.* PhD thesis, Tech. Univ. of Munich, 2002.

[97] L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL*, 2006.

[98] H. Seidl. Deciding equivalence of finite tree automata. *SIAM J. Comput.*, 19(3), 1990.

[99] G. Sénizergues. L(A) = L(B)? Decidability results from complete formal systems. *Theor. Comput. Sci.*, 251(1-2), 2001.

[100] O. Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* PhD thesis, CMU, 1991.

[101] I. D. B. Stark. *Names and Higher-Order Functions.* PhD thesis, Univ. of Cambridge, 1995.

[102] R. Statman. The typed λ-calculus is not elementary recursive. *Theor. Comput. Sci.*, 9, 1979.

[103] C. Stirling. Decidability of DPDA equivalence. *Theor. Comput. Sci.*, 255(1-2), 2001.

[104] C. Stirling. Deciding DPDA equivalence is primitive recursive. In *ICALP*, 2002.

[105] M. Veanes and N. Bjørner. Symbolic automata: The toolkit. In *TACAS*, 2012.

[106] M. Veanes, N. Bjørner, and L. de Moura. Symbolic automata constraint solving. In *LPAR (Yogyakarta)*, 2010.

[107] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *ICST*, 2010.

[108] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, 1993.

[109] D. N. Xu, S. Peyton Jones, and K. Claessen. Static contract checking for Haskell. *SIGPLAN*, 44(1), 2009.

[110] H. Yang. Relational separation logic. *Theor. Comput. Sci.*, 375(1-3), 2007.

Index