Functions with local state: regularity and undecidability

Andrzej S. Murawski

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK

Abstract

We study programs of a finitary ML-like language $\mathsf{RML_f}$ with ground-type references. $\mathsf{RML_f}$ permits the use of functions with locally declared variables that remain private and persist from one use of the function to the next. Using game semantics we show that this leads to undecidability of program equivalence already at second order. We also examine the extent to which this feature can be captured by regular languages. This gives a decidability result for a second-order fragment $\mathsf{RML_f^-}$ of $\mathsf{RML_f}$, which comprises many examples studied in the literature.

Key words: game semantics, automata, software verification, model checking

1 Introduction

Game semantics has contributed fully abstract models for a variety of programming languages. Each of these models gives a semantic account of program equivalence: the interpretations of programs coincide if and only if the programs are equivalent in the respective languages. This makes it possible, at least in principle, to reason about program equivalence with the help of game models. However, their structure does not facilitate such reasoning, since the game categories are obtained via non-trivial quotienting. Fortunately, for languages with ground-type references, the quotient can be characterized explicitly via sets of special plays, making the model more accessible and usable.

Plays in game semantics are sequences of moves equipped with pointers. In some cases, however, the pointer structure can be shown to be uniquely reconstructible and consequently can be ignored. Then sets of plays can be regarded simply as languages over the alphabet of moves and program equivalence can be analyzed as language equivalence. If the associated language equivalence is decidable, so must be program equivalence. In order to obtain decidability, finitary language fragments are considered: with finite datatypes and with iteration instead of unrestricted recursion.

The first result establishing that pattern of reasoning has been obtained by Ghica and McCusker [1,2] and concerned second-order Idealized Algol and regular expressions. This discovery initiated research into algorithmic aspects of game semantics and its potential to become a foundation for software model-checking [3]. The initial decidability result has also been extended in various directions: to third-order Idealized Algol [4,5], concurrency [6] and a call-by-value language with arrays [7]. The last of these papers, by Ghica, investigated a language with first-order procedures and block-allocated variables, such as those used in imperative programs.

In this paper we consider the call-by-value case as well but we shall focus on dynamically-allocated (integer-valued) variables used in languages such as Standard ML. Access to such variables can be passed outside their original allocation block, which opens up new ways of manipulating the program state. One can also define functions which have "private" local variables that persist over invocations and accumulate information throughout their lifetime, like in the simple example given below

$$(\lambda l^{\mathsf{int\,ref}}.\ (\lambda x^{\mathsf{int}}.\mathsf{if}\ !l < \mathtt{max}\ \mathsf{then}\ (l := !l+1; x)\ \mathsf{else}\ \Omega))(\mathsf{ref}(0)) : \mathsf{int} \to \mathsf{int}$$

where the local variable restricts the number of function calls and causes divergence after `max` uses (see also Examples 3, 21, 27 and 28). This encapsulates the state within the function much like in object-oriented programming. Indeed, that mechanism can be employed to define objects and implement basic object-oriented features [8]. The combination of imperative and functional features present in ML turns out quite difficult to reason about [9]. In fact we are going to show that already at second order finitary program equivalence is undecidable. On the other hand, we will identify a language fragment with second-order procedures which can be captured via regular languages and which still contains many examples considered in the literature [9,10]. The language will include some terms whose game semantics is not strictly regular. Then, instead of the full semantics, we will use a suitable regular representative.

Our language of study is finitary RML for which a fully abstract game model was given by Abramsky and McCusker [11]. RML bears close resemblance to Reduced ML as studied by Pitts and Stark [9,10] with one important distinction. RML is equipped with a variable constructor mkvar, which can be used to design user-defined variable objects that do not have to behave like standard memory cells. In general this feature makes RML contexts more discriminating as far as program equivalence is concerned, but this happens only when the types of the terms involved have negative[1] occurrences of $\mathsf{int\,ref}$. In other cases RML can simply be considered a conservative extension of Reduced ML and then our results are immediately applicable to Reduced ML, including the undecidability result which does not make use of mkvar. Recent advances in game semantics suggest that it will soon be possible to construct a fully-abstract game model without using mkvar, using ideas which can be traced back to Fraenkel-Mostowski set theory [12,13]. For the time being, the RML game model can be seen as a simpler and in many cases faithful alternative. In fact for $\mathsf{int\,ref}$-free types the model in [12] coincides with the model for RML.

Outline of the paper. We introduce finitary RML (referred to as RML_f) in Section 2 along with the induced notions of program approximation and equivalence and discuss its relationship to Reduced ML. In Section 3 we give a self-contained description of the game model for RML_f [11]. Since there are no tutorial introductions to call-by-value games, we aim to present the constructions underlying the model in a way that should be understandable to anyone who has read the call-by-name games tutorial [14]. Section 4 contains a systematic study of cases in which pointers are redundant in plays. This motivates the introduction of $\mathsf{RML}_\mathrm{f}^-$ in Section 5, which is a fragment of RML_f that will turn out representable via regular languages. $\mathsf{RML}_\mathrm{f}^-$ supports second-order procedures and makes it possible to encapsulate the state in tuples of first-order functions. Thus, for example, classes with first-order methods can be modelled. Finally, Section 6 presents a reduction of the halting problem to second-order RML_f equivalence.

Related work. A detailed analysis of the behaviour of languages with the features discussed here has been carried out by Pitts and Stark [9,10]. Call-by-value game semantics has been introduced by Honda and Yoshida [15] (in the functional case) and by Abramsky and McCusker [16]. In the context of algorithmic game semantics, this paper is an extension of Ghica's work on regular languages for first-order call-by-value [7]. As for undecidability, the present author has previously shown that fifth-order RML_f with block-allocated variables is undecidable [17,18]. That result was derived from an undecidability result for fourth-order finitary Idealized Algol and did not rely on ML-references. Using a similar approach we show that due to their presence undecidability already occurs at second order.

[1] i.e. in the left-hand scope of a odd number of arrows

$$\frac{}{\Gamma \vdash () : \mathsf{unit}} \qquad \frac{}{\Gamma \vdash \Omega_{\mathsf{unit}} : \mathsf{unit}} \qquad \frac{n \in \{0, \cdots, N\}}{\Gamma \vdash n : \mathsf{int}}$$

$$\frac{}{\Gamma, x : \theta \vdash x : \theta} \qquad \frac{\Gamma \vdash M : \mathsf{int} \qquad \Gamma \vdash M_i : \theta \qquad i = 0, \cdots, N}{\Gamma \vdash \mathsf{case}_\theta(M)[M_0, \cdots, M_N] : \theta}$$

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

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

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

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

Fig. 1. Syntax of $\mathsf{RML_f}$.

2 $\mathsf{RML_f}$

$\mathsf{RML_f}$ is an ML-like language with ground-type references. Its types are generated according to the grammar

$$\theta ::= \quad \mathsf{unit} \quad | \quad \mathsf{int} \quad | \quad \mathsf{int\,ref} \quad | \quad \theta \to \theta,$$

where unit is the type of commands, int is a finite ground datatype corresponding to an initial segment $\{0, \cdots, N\}$ $(N > 0)$ of the set of natural numbers and $\mathsf{int\,ref}$ is the type of int-valued references. $\mathsf{RML_f}$ typing judgments are presented in Figure 1 [2]. The term Ω_{unit} is a "divergent constant" to which no reduction rule will apply. The operational semantics will be given in terms of stores. Let L range over finite sets of *locations* which are taken from some countable set. An L-store s is simply a partial function from L to $\{0, \cdots, N\}$. We write $s(l \mapsto n)$ for the store obtained by updating s so that l is mapped to n (this may extend the domain of s). We say that V is in canonical form if it is either a variable, a location, an int-constant, the unit value (), a λ-abstraction or $\mathsf{mkvar}(\lambda x.M_1)(\lambda x.M_2)$. The operational semantics given in Figure 2 takes the form of judgments

$$\langle\, L, s \,\rangle M \Downarrow \langle\, L', s' \,\rangle V$$

[2] The order of a type is defined by: $ord(\mathsf{unit}) = ord(\mathsf{int}) = 0$, $ord(\mathsf{int\,ref}) = 1$, $ord(\theta_1 \to \theta_2) = \max(1 + ord(\theta_1), ord(\theta_2))$. A term-in-context $\Gamma \vdash M : \theta$ is of order i iff $ord(\theta) \leq i$ and the order of each type in Γ is strictly smaller than i.

where s is an L-store, s' is an L'-store, $L \subseteq L'$ and V is a canonical form. Most rules are presented as

$$\frac{M_1 \Downarrow V_1 \quad M_2 \Downarrow V_2 \quad \cdots \quad M_n \Downarrow V_n}{M \Downarrow V}$$

which is an abbreviation for

$$\frac{\begin{array}{c}\langle L_1, s_1 \rangle\, M_1 \Downarrow \langle L_2, s_2 \rangle\, V_1 \\ \langle L_2, s_2 \rangle\, M_2 \Downarrow \langle L_3, s_3 \rangle\, V_2 \\ \vdots \\ \langle L_n, s_n \rangle\, M_n \Downarrow \langle L_{n+1}, s_{n+1} \rangle\, V_n\end{array}}{\langle L_1, s_1 \rangle M_1 \Downarrow \langle L_{n+1}, s_{n+1} \rangle V}.$$

Note that this means that the ordering of the hypotheses is significant. For a closed term $M : \mathsf{unit}$ we write $M \Downarrow$ iff $\langle \emptyset, \emptyset \rangle M \Downarrow \langle L, s \rangle\, ()$ for some L, s.

Definition 1 Two terms-in-context $\Gamma \vdash M_1$ and $\Gamma \vdash M_2 : \theta$ are *observationally equivalent* ($\Gamma \vdash M_1 \cong M_2$) if for any context $\mathcal{C}[-]$ such that $\mathcal{C}[M_1], \mathcal{C}[M_2]$ are closed terms of type unit, we have $\mathcal{C}[M_1] \Downarrow$ if and only if $\mathcal{C}[M_2] \Downarrow$. Similarly, M_1 *approximates* M_2 ($\Gamma \vdash M_1 \precsim M_2$) if for all contexts satisfying the same properties as above whenever $\mathcal{C}[M_1] \Downarrow$ we also have $\mathcal{C}[M_2] \Downarrow$.

Remark 2 In order to simplify future case analyses we have defined $\mathsf{RML_f}$ using as few syntactic constructs as possible. Because int is finite, the successor, predecessor, addition, subtraction (mod N), $<$, $>$, and $=$ (for int) can all be defined using case_θ. In a similar vein, $\mathsf{if}\, M\, \mathsf{then}\, M_1\, \mathsf{else}\, M_0$ can be defined by $\mathsf{case}_\theta(M)[M_0, M_1, \cdots, M_1]$. Using Ω_{unit} it is possible to define divergent terms Ω_θ at any type.

It is worth noting that the definitions of $\cong$ and $\precsim$ do not depend on the presence of while-loops in contexts, because already without them all compact elements of the game model are definable [19]. However, since while-loops are a common programming idiom, it is good to have them in the language. Actually, their inclusion does not affect our results on (un)decidability. The decidability results will include while, whereas the undecidability argument is carried out without them.

Notation. Throughout the paper we will use let-expressions as syntactic sugar for function application: $\mathsf{let}\, x = M\, \mathsf{in}\, N$ will stand for $(\lambda x.N)M$. For instance, $\mathsf{let}\, x = \mathsf{ref}(0)\, \mathsf{in}\, N$ introduces a location that can range over multiple uses of N. An important special case is when x does not occur in N (for instance, in order to evaluate M to produce side-effects). Then we write $M; N$ or $\mathsf{let}\, \Box = M\, \mathsf{in}\, N$.

$$\frac{}{V \Downarrow V} \qquad \frac{M \Downarrow i \quad M_i \Downarrow V}{\mathsf{case}(M)[\cdots, M_i, \cdots] \Downarrow V}$$

$$\frac{M \Downarrow 0}{\mathsf{while}\, M\, \mathsf{do}\, N \Downarrow ()} \qquad \frac{M \Downarrow i \quad N \Downarrow () \quad \mathsf{while}\, M\, \mathsf{do}\, N \Downarrow ()}{\mathsf{while}\, M\, \mathsf{do}\, N \Downarrow ()} \quad i \neq 0$$

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

$$\frac{\langle L, s \rangle\, M \Downarrow \langle L', s' \rangle n}{\langle L, s \rangle\, \mathsf{ref}\, M \Downarrow \langle L' \cup \{l\}, s'(l \mapsto n) \rangle\, l} \quad l \notin L'$$

$$\frac{\langle L, s \rangle\, M \Downarrow \langle L', s' \rangle\, l \quad s'(l) = n}{\langle L, s \rangle\, !M \Downarrow \langle L', s' \rangle\, n}$$

$$\frac{\langle L, s \rangle\, M \Downarrow \langle L', s' \rangle\, l \quad \langle L', s' \rangle\, N \Downarrow \langle L'', s'' \rangle n}{\langle L, s \rangle\, M := N \Downarrow \langle L'', s''(l \mapsto n) \rangle\, ()}$$

$$\frac{M \Downarrow V_1 \quad N \Downarrow V_2}{\mathsf{mkvar}\, M\, N \Downarrow \mathsf{mkvar}\, V_1\, V_2}$$

$$\frac{M \Downarrow \mathsf{mkvar}\, V_1\, V_2 \quad V_1() \Downarrow n}{!M \Downarrow n} \qquad \frac{M \Downarrow \mathsf{mkvar}\, V_1\, V_2 \quad N \Downarrow n \quad V_2(n) \Downarrow ()}{M := N \Downarrow ()}$$

Fig. 2. Operational Semantics of $\mathsf{RML_f}$

Additionally, $\mathsf{let}\, x_1 = M_1, \cdots, x_n = M_n\, \mathsf{in}\, M$ will abbreviate the obvious nesting of n let-constructs. We will also use "assertions" of the shape [*condition*] which are shorthand for if *condition* then () else Ω_{unit}.

Example 3 $\mathsf{let}\, v = \mathsf{ref}(0)\, \mathsf{in}\, \lambda x.((v := x + !v); !v)$ keeps a running total of all arguments it has been applied to. This should be contrasted with $\lambda x.\mathsf{let}\, v = \mathsf{ref}(0)\, \mathsf{in}\, ((v := x + !v); !v)$, which creates a new copy of v for each call in which it stores the current argument. The latter term, but not the former, could be defined in a call-by-value Algol-like language.

The syntax and operational semantics of $\mathsf{RML_f}$ are faithful to Standard ML with one notable exception. $\mathsf{RML_f}$ has the mkvar-constructor which makes it possible to define variable objects from user-defined read- and write-methods. Such objects, known as "bad variables", rarely behave like genuine variables. For instance, $\mathsf{mkvar}(\lambda x.1, \lambda x.\Omega_{\mathsf{unit}})$ always returns 1 when dereferenced and diverges at the first write request. mkvar is present in $\mathsf{RML_f}$ for semantic reasons, without it the game model in [11] would lose the definability property (any compact strategy is definable by a term) and would not be fully abstract.

The canonical restriction of Standard ML to ground-type references (without bad variables), called Reduced ML, has been introduced in [10] and studied in [9]. Any (finitary) Reduced ML term can be viewed as an $\mathsf{RML_f}$ term: the only obstacle is the equality test $eq : \mathsf{int\,ref} \to \mathsf{int\,ref} \to \mathsf{int}$ for locations, but it is not a primitive operation and can be replaced, for instance, by $\lambda x.\lambda x'.\mathsf{let}\, v = !x,\ b = (x := !x' + 1; (!x = !x'))\ \mathsf{in}\, (x := v; b)$ [9]. Consequently, one can analyze (finitary) Reduced ML terms in $\mathsf{RML_f}$ and ask when $\mathsf{RML_f}$ is a conservative extension of (finitary) Reduced ML. Clearly, due to the possible bad-variable behaviour, $\mathsf{RML_f}$-contexts are more general, so we have:

Proposition 4 *Let $\Gamma \vdash M_1, M_2 : \theta$ be Reduced ML terms. Then $M_1 \lesssim_{\mathsf{RML_f}} M_2$ implies $M_1 \lesssim_{\mathrm{ReducedML}} M_2$ (and similarly for $\cong$).*

The converse does not hold in general: for instance, $\lambda v^{\mathsf{int\,ref}}.(v := !v) \cong \lambda v.()$ holds in Reduced ML, but not in $\mathsf{RML_f}$, because the terms might be applied to the bad variable created above, which would result in divergence for the left term. Note that these terms have type $\theta = \mathsf{int\,ref} \to \mathsf{unit}$, i.e. $\mathsf{int\,ref}$ has a negative occurrence. In fact this characterizes the cases when Reduced ML and $\mathsf{RML_f}$ differ. If θ has no negative occurrences of $\mathsf{int\,ref}$, then the type of the context, namely $\theta \Rightarrow \mathsf{unit}$, has no positive occurrences of $\mathsf{int\,ref}$. For such types, the definability proof for compact strategies does not need mkvar [19,11]: any finite strategy on $\theta \Rightarrow \mathsf{unit}$ is definable by a mkvar-free term. Consequently, for any discriminating $\mathsf{RML_f}$ context we can find one without mkvar.

Proposition 5 *Let θ be an $\mathsf{RML_f}$-type without negative occurrences of $\mathsf{int\,ref}$. Let $\vdash M_1, M_2 : \theta$ be Reduced ML terms. Then $M_1 \lesssim_{ReducedML} M_2$ implies $M_1 \lesssim_{\mathsf{RML_f}} M_2$. Consequently, $\mathsf{RML_f}$ is a conservative extension of Reduced ML at type θ: $M_1 \lesssim_{ReducedML} M_2 \iff M_1 \lesssim_{\mathsf{RML_f}} M_2$.*

The proposition can readily be extended to open terms using the fact that $\Gamma \vdash M_1 \lesssim M_2$ is equivalent to $\vdash \lambda\Gamma.M_1 \lesssim \lambda\Gamma.M_2$. Hence, facts proved about $\mathsf{RML_f}$ will often carry over to Reduced ML. This will be the case with both the decidability and the undecidability results we are about to present. It is worth stressing that the restriction on the occurrences of $\mathsf{int\,ref}$ concerns only the types of the terms M_1, M_2. Their typing derivations might well contain negative occurrences of $\mathsf{int\,ref}$, but these have to be bound in later stages of the derivation.

3 Game Semantics of $\mathsf{RML_f}$

Game semantics views types as games. The games involve two players, called Opponent (O) and Proponent (P), who make their moves alternately. Terms are then interpreted as strategies for Proponent. Call-by-value game semantics [19] fits into the framework proposed by Moggi in [20], where he shows how, given a cartesian-closed category equipped with a strong monad, one can model the call-by-value lambda calculus. In the case of games, the underlying cartesian-closed category can be constructed through the "families construction" (known as free coproduct completion in category theory) and the monadic action is defined using sum games. Below we present all the elements in a rather direct way aiming to bridge the gap between the abstract original presentation in [11] and the concrete representation we are aiming to describe later for a fragment of $\mathsf{RML}_{\mathrm{f}}$. Most of the definitions that follow are by now standard in game semantics. We include them to ensure a self-contained presentation of the call-by-value model. For detailed proofs the reader should consult [21].

3.1 Games and strategies

Each game has an underlying arena which is used to specify the combinatorial constraints that moves by O and P must satisfy.

Definition 6 An *arena* A is a triple $\langle M_A, \lambda_A, \vdash_A \rangle$, where

- M_A is a set of moves;
- $\lambda_A : M_A \to \{O, P\} \times \{Q, A\}$ is a function determining for each $m \in M_A$ whether it is an *Opponent* or a *Proponent* move and whether it is a *question* or an *answer* (we write $\lambda_A^{OP}, \lambda_A^{QA}$ for the composite of λ_A with respectively the first and second projection);
- $\vdash_A$ is a relation between $M_A + \{\diamond\}$ and M_A, called *enabling*, satisfying the three conditions below:

(1) if $\diamond \vdash_A n$, then $\lambda_A(n) = (O, Q)$ and $m \not\vdash_A n$ for any $m \in M_A$,
(2) if $m \vdash_A n$ and $m \neq \diamond$ then $\lambda_A^{OP}(m) \neq \lambda_A^{OP}(n)$,
(3) if $m \vdash_A n$ and $\lambda_A^{QA}(n) = A$ then $\lambda_A^{QA}(m) = Q$.

We shall write I_A for the set of all moves of A which are enabled by $\diamond$; such moves are called *initial*. By (1) an initial move must be an Opponent question. If $m \vdash_A n$ we say that m *enables* n. By (2) players always enable each other's moves, never their own and, by (3), answers can only be enabled by questions.

Not all sequences of moves are regarded as legal in the games. The legal positions are defined below in a series of definitions: in short, they are justified sequences satisfying the visibility and bracketing conditions.

Definition 7 A *justified sequence* in arena A is a finite sequence of moves of A equipped with pointers: each occurrence of a non-initial move n must have a unique pointer to an earlier occurrence of a move m such that $m \vdash_A n$. We then say that n is (explicitly) justified by m and, when n is an answer, that n answers m.

We shall also say that n is *hereditarily justified* by m is there is a chain of pointers leading from n back to m. We will write $s \upharpoonright m$ for the subsequence of s consisting of moves hereditarily justified by m.

For a justified sequence s, the O-view $\llcorner s \lrcorner$ of s and the P-view $\ulcorner s \urcorner$ of s are defined as follows.

$$
\begin{array}{rclcrcl}
\llcorner \epsilon \lrcorner & = & \epsilon & \qquad\qquad & \ulcorner \epsilon \urcorner & = & \epsilon \\
\llcorner s\, o \lrcorner & = & \llcorner s \lrcorner\, o & & \ulcorner s\, p \urcorner & = & \ulcorner s \urcorner\, p \\
\llcorner s\, \overparen{o\, t\, p} \lrcorner & = & \llcorner s \lrcorner\, \overparen{o\, p} & & \ulcorner s\, \overparen{p\, t\, o} \urcorner & = & \ulcorner s \urcorner\, \overparen{p\, o}
\end{array}
$$

The notions of views are used to define legal sequences next.

Definition 8 A justified sequence s is *legal* if and only if

- The players alternate: if $s = s_1 m n s_2$ then $\lambda^{OP}(m) \neq \lambda^{OP}(n)$.
- The *bracketing* condition holds: whenever an answer move is played, it answers the most recent unanswered question.
- The *visibility* condition holds: whenever $s = s_1 m s_2 n s_3$ and n is explicitly justified by m then m must occur in $\llcorner s_1 m s_2 \lrcorner$ if n is an O-move or in $\ulcorner s_1 m s_2 \urcorner$ if n is a P-move.

Games are defined by specifying the legal positions that can be explored by the two players.

Definition 9 A *game* A is specified by a structure $\langle M_A, \lambda_A, \vdash_A, P_A \rangle$ where $\langle M_A, \lambda_A, \vdash_A \rangle$ is an arena and P_A is a non-empty, prefix-closed subset of L_A, called the *positions* of A, satisfying the following condition: if $s \in P_A$ and i is an initial move in s, then $s \upharpoonright i \in P_A$.

The simplest game is the empty game $\langle \emptyset, \emptyset, \emptyset, \{\epsilon\} \rangle$, which will be denoted by 1. In Figure 3 we present several standard constructions on games that are needed to interpret RML_f-types. $+$ on the right-hand side of the defining equations denotes the disjoint sum of two sets, we write $\sum_{i \in I} X_i$ if the disjoint sum of a family of sets is involved.

The definitions of Figure 3 imply the so-called *switching conditions*. Each play in the $A \otimes B$ game is an interleaving of a play of A with a play of B, but it is only O who can switch between them. In $A \multimap B$ the ownership of moves in A is reversed. Plays are also interleavings of plays from the component games, but now only P can switch between them. In $A \times B$ each play proceeds entirely in A or in B, no interleavings occur. For a change, the set of plays on $!A$ consists of any number of interleavings of plays in A. Finally, in $\sum_{i \in I} A_i$ O has to start with q, after which P "chooses" $i \in I$; afterwards the play proceeds as in A_i.

$$
\begin{aligned}
M_{A\otimes B} &= M_A + M_B \\
\lambda_{A\otimes B} &= [\lambda_A, \lambda_B] \\
\vdash_{A\otimes B} &= \vdash_A + \vdash_B \\
P_{A\otimes B} &= \{s \in L_{A\otimes B} \mid s \upharpoonright A \in P_A \ \wedge \ s \upharpoonright B \in P_B\}
\end{aligned}
$$

$$
\begin{aligned}
M_{A\multimap B} &= M_A + M_B \\
\lambda_{A\multimap B} &= [\langle\, \lambda_A^{PO}, \lambda_A^{QA}\, \rangle, \lambda_B] \\
\vdash_{A\multimap B} &= (\vdash_A \cap (M_A \times M_A)) \ + \ \{(b,a) \mid b \in I_B \ \wedge \ a \in I_A\} + \ \vdash_B \\
P_{A\multimap B} &= \{s \in L_{A\multimap B} \mid s \upharpoonright A \in P_A \ \wedge \ s \upharpoonright B \in P_B\}
\end{aligned}
$$

$$
\begin{aligned}
M_{A\times B} &= M_A + M_B \\
\lambda_{A\times B} &= [\lambda_A, \lambda_B] \\
\vdash_{A\times B} &= \vdash_A + \vdash_B \\
P_{A\times B} &= \{s \in L_{A\times B} \mid s \upharpoonright A \in P_A \ \wedge \ s \upharpoonright B = \epsilon\} \cup \\
&\quad\ \{s \in L_{A\times B} \mid s \upharpoonright A = \epsilon \ \wedge \ s \upharpoonright B \in P_B\}
\end{aligned}
$$

$$
\begin{aligned}
M_{!A} &= M_A \\
\lambda_{!A} &= \lambda_A \\
\vdash_{!A} &= \vdash_A \\
P_{!A} &= \{s \in L_{!A} \mid \text{for all } m \in I_A,\ s \upharpoonright m \in P_A\}
\end{aligned}
$$

$$
\begin{aligned}
M_{\sum_{i\in I} A_i} &= \{q\} + I + \textstyle\sum_{i\in I} M_{A_i} \\
\lambda_{\sum_{i\in I} A_i} &= \{(q,O)\} + (I \times \{P\}) + \textstyle\sum_{i\in I} \lambda_{A_i} \\
\vdash_{\sum_{i\in I} A_i} &= \{(\diamond, q)\} \ + \ \{(q,i) \mid i \in I\} + \{(i,m) \mid i \in I,\ m \in I_{A_i}\} + \\
&\quad\ \textstyle\sum_{i\in I} (\vdash_{A_i} \cap (M_{A_i} \times M_{A_i})) \\
P_{\sum_{i\in I} A_i} &= \{\epsilon, q\} \cup \{qi \mid i \in I\} \cup \{qis \mid i \in I, s \in P_{A_i}\}
\end{aligned}
$$

$s \upharpoonright A$, $s \upharpoonright B$ denote the subsequence of s consisting of all moves from M_A and M_B respectively. λ_A^{PO} is defined by: $\lambda_A^{PO}(m) = O$ if and only if $\lambda_A^{OP}(m) = P$.

Fig. 3. Game constructions

Notation. Most of the game constructions rely on the disjoint sum. When we give examples of positions we will use subscripts and superscripts to indicate various occurrences of the same move. For instance, if two copies of the same game occur on the left and right side of $\vdash$ we normally use the subscripts l, r. Similarly, we use the superscripts i for moves originating from components of $\sum_{i\in I} X_i$. We shall often treat the disjoint sum as associative and commutative without mentioning the associated isomorphisms explicitly.

Games are the objects of game categories. Strategies will be their morphisms. We consider deterministic strategies only.

Definition 10 A *strategy* σ for a game A is a non-empty set of even-length positions from P_A, satisfying:

- if $sab \in \sigma$ then $s \in \sigma$;
- if $sab, sac \in \sigma$ then $b = c$ (the justifiers of b and c are also required to be the same so that we have $sab = sac$).

The *identity* strategy for $A \multimap A$ is defined by

$$\mathsf{id}_A = \{s \in P^{\text{even}}_{A_1 \multimap A_2} \quad | \quad \forall t \sqsubseteq^{\text{even}} s. \quad t \upharpoonright A_1 = t \upharpoonright A_2\}$$

where P^{even}_A is the subset of P_A consisting of even-length positions, $t \sqsubseteq^{\text{even}} s$ means that t is an even-length prefix of s and the subscripts $1, 2$ distinguish the two occurrences of A. id_A simply copies Opponent moves to the other copy of A.

Two strategies $\sigma : A \multimap B$ and $\tau : B \multimap C$ can be composed (to yield a strategy $\sigma; \tau : A \multimap C$) by considering their possible interactions in the shared game B; moves in B are subsequently hidden yielding a sequence of moves in A and C. More formally, let u be a sequence of moves from arenas A, B and C with justification pointers from all moves except those initial in C such that pointers from moves in C cannot point to moves in A and vice versa. Define $u \upharpoonright B, C$ to be the subsequence of u consisting of all moves from B and C (pointers between A-moves and B-moves are ignored). $u \upharpoonright A, B$ is defined analogously (pointers between B and C are then ignored). 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 all such sequences is written as $\mathsf{int}(A, B, C)$. Then we let

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

where $u \upharpoonright A, C$ is the subsequence of u consisting of all moves from A and C, but where there was a pointer from a move $m_A \in M_A$ to an initial move $m_B \in M_B$ we extend the pointer to the initial move in C which was pointed to from m_B.

3.2 Categories of Games

Games and strategies form a symmetric monoidal closed category in which morphisms between A and B are given by strategies on $A \multimap B$ composed as defined above. The $\otimes$ construction is the requisite tensor product and $\multimap$ provides the necessary closure. The empty game is the tensor unit. This category is a stepping stone towards a cartesian-closed category which will be defined next. For that it is necessary to consider a restricted class of games.

Definition 11 A game A is *well-opened* iff for all $sm \in P_A$, where m is initial, we have $s = \epsilon$.

Given $\sigma :!A \multimap B$, where B is well-opened, we can define $\sigma^\dagger :!A \multimap !B$ by interleaving plays from σ. The purpose of the restriction to well-opened games is the fact that σ is determined uniquely by $\sigma^\dagger$ only if B is well-opened.

Now the cartesian-closed category $\mathcal{I}$ is defined by taking the objects to be well-opened games and taking morphisms between A and B to be strategies for the (well-opened) game $!A \multimap B$. $\sigma :!A \multimap B$ and $\tau :!B \multimap C$ are composed by taking $\sigma^\dagger; \tau :!A \multimap C$. The identities in $\mathcal{I}$ are given by strategies $\mathsf{der}_A :!A \multimap A$ which are simply retyped variants of $\mathsf{id}_A : A \multimap A$. From now on we will restrict our attention to $\mathcal{I}$, so whenever we use ; and id_A for strategies we mean the composition and the identities in $\mathcal{I}$.

Theorem 12 ([21,19]) *$\mathcal{I}$ is a cartesian-closed category with products given by $\times$ and function spaces $A \Rightarrow B$ taken to be $!A \multimap B$.*

Projections in $\mathcal{I}$ are essentially the same as identity strategies on $\mathcal{I}$, embedded into the game $A \times B \Rightarrow A$. Pairing amounts to taking the disjoint sum of two strategies, whereas currying is essentially the identity operation since $A \times B \Rightarrow C$ is isomorphic to $A \Rightarrow (B \Rightarrow C)$. The product construction can easily be generalized to give small products $\prod_{i \in I} A_i$. The empty game 1 is the terminal object. In fact A, $1 \times A$, $A \times 1$, $1 \Rightarrow A$ are all identical up to the embedding into the disjoint sum. $\mathcal{I}$ will be the base category over which another cartesian-closed category *Fam*($\mathcal{I}$) is constructed. *Fam*($\mathcal{I}$) will be the category in which call-by-value can be interpreted following Moggi's recipe [20].

The objects of *Fam*($\mathcal{I}$) are families of games $\{A_i\}_{i \in I}$ indexed by a set I. A *Fam*($\mathcal{I}$)-morphism from $\{A_i\}_{i \in I}$ to $\{B_j\}_{j \in J}$ is a pair $(f, \{\sigma_i\}_{i \in I})$ where $f : I \to J$ is a function and $\sigma_i : A_i \to B_{f(i)}$ is a $\mathcal{I}$-morphism. These are composed componentwise: given $(f, \{\sigma_i\}_{i \in I}) : \{A_i\}_{i \in I} \to \{B_j\}_{j \in J}$ and $(g, \{\tau_j\}_{j \in J}) : \{B_j\}_{j \in J} \to \{C_k\}_{k \in K}$, $(f, \{\sigma_i\}); (g, \{\tau_j\})$ is defined to be $(g \circ f, \{\nu_i\}_{i \in I})$ where $\nu_i = \sigma_i; \tau_{f(i)} : A_i \to C_{g(f(i))}$. The identity morphisms are given by identity functions along with families of identity strategies (on $\mathcal{I}$).

Fam($\mathcal{I}$) has all small products, which are calculated pointwise, e.g.

$$\{A_i\}_{i \in I} \times \{B_j\}_{j \in J} \;=\; \{A_i \times B_j \mid (i, j) \in I \times J\}.$$

Projections are defined by set-theoretic projections along with projections taken from $\mathcal{I}$. Similarly, pairing is defined by pairing up functions and strategies (as in $\mathcal{I}$). The terminal object is the singleton family with the empty game. Exponentials in $Fam(\mathcal{I})$ are defined by

$$\{A_i \mid i \in I\} \Rightarrow \{B_j \mid j \in J\} \;=\; \{\prod_{i\in I}(A_i \Rightarrow B_{f(i)}) \mid f \in J^I\}.$$

Given a map $(f, \{\sigma_{i,j}\}_{(i,j)\in I\times J}) : \{A_i\}_{i\in I}\times\{B_j\}_{j\in J} \to \{C_k\}_{k\in K}$ the corresponding curried map $\Lambda(f, \{\sigma_{i,j}\})$ is $(\Lambda(f), \{\tau_i\}_{i\in I})$ where $\tau_i : A_i \to \prod_{j\in J}(B_j \Rightarrow K_{f(i,j)})$ is obtained by J-tupling the family $\{\Lambda(\sigma_{i,j}) : A_i \to (B_j \Rightarrow C_{f(i,j)})\}_{j\in J}$ in $\mathcal{I}$.

Theorem 13 ([11]) *$Fam(\mathcal{I})$ is a cartesian-closed category.*

The sum-game construction can be used to define a monad T on $Fam(\mathcal{I})$. Given a family $\{A_i\}_{i\in I}$ of games, we let $T(\{A_i\}_{i\in I})$ be the singleton family (indexed by $\{\star\}$) containing $\sum_{i\in I}!A_i$. Recall that each play of $\sum_{i\in I}!A_i$ begins with q played by O, after which P plays $i \in I$. What follows is an interleaving of an arbitrary number of plays in A_i (only O can switch between the interleavings). Given a morphism $(f, \{\sigma_i\}) : \{A_i\}_{i\in I} \to \{B_j\}_{j\in J}$, $T(f, \{\sigma_i\}) : T(\{A_i\}) \to T(\{B_j\})$ is defined to be $(g, \{\sigma\})$ where $g : \{\star\} \to \{\star\}$ is the unique endofunction on $\{\star\}$ and $\sigma : \sum_{i\in I}!A_i \to \sum_{j\in J}!B_j$ is the smallest strategy containing plays of the shape $q_r\, q_l\, i_l\, f(i)_r\, s$ where $s \in \sigma_i^\dagger$. The monad unit $\eta_{\{A_i\}_{i\in I}} : \{A_i\}_{i\in I} \to T(\{A_i\}_{i\in I})$ is defined by $(f, \{\eta_i\})$ where $f : I \to \{\star\}$ is the unique function from I to $\{\star\}$, and $\eta_i : A_i \Rightarrow \sum_{i\in I}!A_i$ are strategies which tell P to reply with i to the initial q and thereafter copy moves of A_i between its left and right copies, as in $\mathsf{id}^\dagger_{A_i}$.

T can be shown to be a *strong monad* (Definition 2.2 in [20]) which implies the existence of a family of the so-called (left) partial-pairing morphisms $\mathsf{left}_{A,B} : T(A)\times T(B) \to T(A\times B)$ that play a vital role in defining the interpretations of call-by-value terms. In $Fam(\mathcal{I})$ the maps $\mathsf{left}_{\{A_i\}_{i\in I},\{B_j\}_{j\in J}}$ are given by (f, σ) where $f : \{\star\}\times\{\star\} \to \{\star\}$ is the obvious function and σ has plays as shown below. Note that below and in what follows we write $T(\{A_i\}_{i\in I})$ for the unique game in $T(\{A_i\}_{i\in I})$.

$$\begin{array}{lccccc}
 & T(\{A_i\}_{i\in I}) & \times & T(\{B_j\}_{j\in J}) & \Rightarrow & T(\{A_i\times B_j\}_{(i,j)\in I\times J}) \\
O & & & & & q \\
P & q & & & & \\
O & i & & & & \\
P & & & q & & \\
O & & & j & & \\
P & & & & & (i,j) \\
 & & & \vdots & &
\end{array}$$

where $\vdots$ denotes an exchange of moves where P simply copies moves between the left and right copies of A_i and B_j (like in $\mathsf{id}^{\dagger}_{(A_i \times B_j)}$).

3.3 CBV game semantics

We have now highlighted all the structure that is needed to model the call-by-value lambda calculus in $\mathit{Fam}(\mathcal{I})$ in the standard way [20]. Below we examine that interpretation in more detail and show how other elements of the $\mathsf{RML_f}$ syntax are interpreted.

Typing derivations $x_1 : \theta_1, \cdots, x_k : \theta_k \ \vdash\ M : \theta$ will be interpreted as $\mathit{Fam}(\mathcal{I})$-morphisms between $[\![\theta_1, \cdots, \theta_k]\!] = [\![\theta_1]\!] \times \cdots \times [\![\theta_k]\!]$ and $T([\![\theta]\!])$. $\mathsf{RML_f}$ types will be interpreted as families of games $\{A_i\}_{i \in I}$. The indexing sets I will be of a rather simple form: I will be either the singleton set $\{\star\}$ or the set $\{0, \cdots, N\}$.

- $[\![\mathsf{unit}]\!]$ is the singleton family $\{1_i\}_{i \in \{\star\}}$.
- $[\![\mathsf{int}]\!]$ is the $\{0, \cdots, N\}$-indexed family of empty games $\{1_i\}_{i \in \{0, \cdots, N\}}$.
- $[\![\mathsf{int\,ref}]\!]$ is the singleton family $\{\mathsf{var}\}_{i \in \{\star\}}$ such that

$$M_{\mathsf{var}} = \{\mathit{read}\} + \{i \mid i = 0, \cdots, N\} + \{\mathit{write}(i) \mid i = 0, \cdots, N\} + \{\mathit{ok}\}$$

  where read and $\mathit{write}(i)$ $(0 \leq i \leq N)$ are initial O-questions, i and ok are P-answers $(0 \leq i \leq N)$ such that $\mathit{read} \vdash_{\mathsf{var}} i$ and $\mathit{write}(i) \vdash_{\mathsf{var}} \mathit{ok}$; plays are prefixes of $(\sum_{i=0}^{N} (\mathit{read}\, i + \mathit{write}(i)\, \mathit{ok}))^*$.
- $[\![\theta_1 \to \theta_2]\!] = [\![\theta_1]\!] \Rightarrow T([\![\theta_2]\!])$. That is, assuming $[\![\theta_1]\!] = \{A_i\}_{i \in I}$, $[\![\theta_1 \to \theta_2]\!]$ will be the singleton family containing $\prod_{i \in I} (A_i \Rightarrow T([\![\theta_2]\!]))$.

Observe that whenever $[\![\theta]\!] = \{B_i\}_{i \in I}$ we actually have $B_i = B_{i'}$ for $i, i' \in I$, so we shall write $\{B\}_{i \in I}$ instead. Suppose $[\![\theta_j]\!] = \{A_j\}_{i_j \in I_j}$ for $j = 1, \cdots, k$. Then $\mathit{Fam}(\mathcal{I})$-morphisms between $[\![\theta_1]\!] \times \cdots \times [\![\theta_k]\!]$ and $T([\![\theta]\!])$ are pairs $(f, \{\sigma_{\vec{i}}\}_{\vec{i} \in \vec{I}})$ where $\vec{I} = I_1 \times \cdots \times I_k$ and $f : \vec{I} \to \{\star\}$. Since there is only one such f, the morphisms are simply $\vec{I}$-indexed families $\{\sigma_{\vec{i}}\}_{\vec{i} \in \vec{I}}$ of strategies where given $\vec{i} = (i_1, \cdots, i_k)$, $\sigma_{\vec{i}} : A_1 \times \cdots \times A_k \Rightarrow T(\{B\}_{i \in I})$. Note that all these strategies are for the same game. Besides, $I_i \neq \{\star\}$ if and only if $\theta_i = \mathsf{int}$. Thus, intuitively, each of the constituent strategies corresponds to interpreting M in which the free variables of type int are replaced with numerical values given by $(i_1, \cdots, i_k)$.

Example 14 Let $\mathit{succ} = \lambda x^{\mathsf{int}}.\mathsf{case}_{\mathsf{int}}(x)[1, \cdots, N, 0]$. Then $x : \mathsf{int} \ \vdash\ \mathit{succ}\, x : \mathsf{int}$ will be interpreted by $\{\sigma_i\}_{i \in \{0, \cdots, N\}}$ such that $\sigma_i = \{\epsilon, q(i+1)\}$. σ_i is a strategy for $1 \Rightarrow T([\![\mathsf{int}]\!])$, i.e. essentially $T([\![\mathsf{int}]\!])$. The term $\vdash\ \lambda x^{\mathsf{int}}.\mathit{succ}\, x$ is then interpreted by a single strategy (strictly speaking, a $\{\star\}$-indexed family of strategies) on $\sum_{j \in \{\star\}} !(\prod_{i \in \{0, \cdots, N\}} T([\![\mathsf{int}]\!]))$ with positions of the shape

$q \star (\sum_{i=0}^{N} q^i (i+1)^i)^*$.

Remark 15 The unique game in $T(\llbracket \mathsf{unit} \rrbracket)$ is the same as the one used for modelling the type of commands in call-by-name languages. For the sake of uniformity, we will use *run*, *done* (or simply r, d) instead of $q, \star$ to refer to its moves. Similarly, the game $T(\llbracket \mathsf{int} \rrbracket)$ is identical to the call-by-name game representing the type of expressions. Note also that $\llbracket \mathsf{int\,ref} \rrbracket$ is essentially the same as $\llbracket \mathsf{unit} \to \mathsf{int} \rrbracket \times \llbracket \mathsf{int} \to \mathsf{unit} \rrbracket$. This highlights the commitment to Reynolds' idea of representing variables as a product of the read and write methods. This view however imposes itself on the syntax and necessitates the introduction of mkvar if one aims for a fully abstract model.

Strategies interpreting typing judgments are for the game $A_1 \times \cdots \times A_k \Rightarrow T(\{B\}_{i \in I})$. Plays on that game are prefixes of sequences matching

$$q \ \underbrace{(M_{A_1} + \ldots + M_{A_k})^*}_{\text{Stage 1}} \ i \ \underbrace{(M_{A_1} + \ldots + M_{A_k} + M_B)^*}_{\text{Stage 2}}.$$

Informally, Stage 1 corresponds to evaluation resulting in the value i. For $T(\llbracket \mathsf{int} \rrbracket)$, i will be one of the numerals, for $T(\llbracket \mathsf{unit} \rrbracket)$ the move *done* signalling termination. For these two types Stage 2 does not happen (there is nothing more to compute). In contrast, for $T(\llbracket \mathsf{int\,ref} \rrbracket)$ further play in $!B$ will correspond to reading and writing. Similarly, for function types, the play in $!B$ will correspond to multiple, possibly nested, invocations with possibly varying arguments.

The constants $()$: unit and $n : \mathsf{int}$ are interpreted by

$$\llbracket \Gamma \vdash () : \mathsf{unit} \rrbracket = \{r\, d\}_{i \in \vec{I}} \qquad \llbracket \Gamma \vdash n : \mathsf{int} \rrbracket = \{q\, n\}_{i \in \vec{I}}$$

where Stage 1 is trivial and Stage 2 does not take place (neither stage occurs for $\llbracket \Gamma \vdash \Omega_\theta \rrbracket = \{\{\epsilon\}\}_{i \in \vec{I}}$). Here is an example of a term for which both stages will be present.

Example 16 The term

$$\begin{array}{c} g : \mathsf{int} \to \mathsf{int} \vdash \mathsf{let}\ f = (\mathsf{if}_{\mathsf{int} \to \mathsf{int}}\, g(1)\, \mathsf{then}\ succ\ \mathsf{else}\ \lambda x. succ(succ\, x)) \\ \mathsf{in}\ \lambda x. \mathsf{if}_{\mathsf{int}}\, g(0)\, \mathsf{then}\ f x\ \mathsf{else}\ x : \mathsf{int} \to \mathsf{int} \end{array}$$

generates the position $q\, q_l^1\, 0_l^1 \star q_r^3\, q_l^0\, 4_l^0\, 5_r^3\, q_r^2\, q_l^0\, 0_l^0\, 2_r^2$.

Now we briefly review all special strategies involved in modelling other features of $\mathsf{RML_f}$. The first one is the monad unit η which is used to interpret free identifiers and λ-abstraction. Suppose $\{\Gamma,\theta\} = \{A_1 \times \cdots \times A_k \times B\}_{(\vec{i},i)\in\vec{I}\times I}$.

- We have $[\![\Gamma, x:\theta \ \vdash\ x:\theta]\!] = \pi; \eta_{\{B\}_{i\in I}}$, where $\pi : [\![\Gamma,\theta]\!] \to [\![\theta]\!]$ is a suitable projection in $\mathit{Fam}(\mathcal{I})$, i.e. $\pi;\eta_{\{B\}_{i\in I}} = \{\sigma_{\vec{i},i}\}_{(\vec{i},i)\in\vec{I}\times I}$, where $\sigma_{(i_1,\cdots,i_k,i)} = \pi_{k+1};\eta_i$, $\pi_{k+1} : A_1\times\cdots\times A_k \times B \Rightarrow B$ is a projection in $\mathcal{I}$ and $\eta_i : B \Rightarrow \sum_{i\in I}!B$.
- We have $[\![\Gamma \ \vdash\ \lambda x.M : \theta\to\theta']\!] = \Lambda([\![\Gamma, x:\theta \ \vdash\ M:\theta']\!]);\eta_{[\![\theta\to\theta']\!]}$, i.e. assuming $[\![\Gamma, x:\theta \ \vdash\ M:\theta']\!] = \{\sigma'_{\vec{i},i}\}_{(\vec{i},i)\in\vec{I}\times I}$ and $[\![\theta\to\theta']\!] = \{C\}$ we have $[\![\Gamma \ \vdash\ \lambda x.M:\theta\to\theta']\!] = \{\sigma_{\vec{i}}\}_{\vec{i}\in\vec{I}}$ where $\sigma_{\vec{i}} = \langle\, \Lambda(\sigma'_{\vec{i},i}) \mid i \in I\,\rangle;\eta_\star$, where $\eta_\star : C \Rightarrow \sum_{i\in\{\star\}}!C$.

Next we consider the remaining rules of the shape

$$\frac{\Gamma \ \vdash\ M_1:\theta_1 \quad \cdots \quad \Gamma \ \vdash\ M_k:\theta_k}{\Gamma \ \vdash\ \mathsf{op}(M_1,\cdots,M_k):\theta}$$

where op represents application or any of case_θ, ref, $:=$, $!$, mkvar, while. Suppose $[\![\Gamma \ \vdash\ M_j]\!] = \{\sigma_i^j\}_{i\in I}$. Then $[\![\Gamma \ \vdash\ \mathsf{op}(M_1,\cdots,M_k)]\!] = \{\sigma_i\}_{i\in I}$ is defined compositionally by $\sigma_i = \langle\,\sigma_i^1,\cdots,\sigma_i^k\,\rangle;\sigma_{\mathsf{op}}$, where σ_{op} is the corresponding special strategy. We analyze all the special strategies in turn.

For application one uses the maps $\mathsf{apply}_{\{B_j\}_{j\in J},\{C_k\}_{k\in K}}$ which are obtained as $\mathsf{left};T(\mathsf{ev})$, where ev is an application map of $\mathit{Fam}(\mathcal{I})$. We describe their shape explicitly below.

$$\begin{array}{lccc}
 & T(\{\prod_{j\in J}(B_j \Rightarrow T(\{C_k\}_{k\in K}))\}) & \times\ T(\{B_j\}_{j\in J}) & \Rightarrow\ T(\{C_k\}_{k\in K}) \\
O & & & q \\
P & q & & \\
O & \star & & \\
P & & q & \\
O & & j & \\
P & q^j & & \\
 & & \vdots &
\end{array}$$

apply first visits the function component (second move). The next move by O is a signal that a function value has been reached, then the strategy proceeds to the argument (fourth move). The next O-move j corresponds to a completed evaluation which triggers q^j, a move which begins the computation of the value of the application by entering the jth component of the product. After that the play consists in copying moves between the two copies of $!B_j$ and $T(\{C_k\}_{k\in K})))$ (only O can switch between the two copying modes). Note that if $\{B_j\}_{j\in J}$ is $[\![\mathsf{int}]\!]$ or $[\![\mathsf{unit}]\!]$ each B_j is an empty game so no further play in $T(\{B_j\}_{j\in J})$ is possible after the fifth move. In contrast, for $\mathsf{int\,ref}$, $!B_j$ is used to copy read and write requests for variables.

For other cases of op the corresponding strategies are the smallest strategies containing the respective positions listed below.

- $\mathsf{case}_\theta : T([\![\mathsf{int}]\!]) \times \prod_{i=0}^{N} T([\![\theta]\!]) \Rightarrow T([\![\theta]\!])$

$$q\, q_{\mathsf{int}} \sum_{i=0}^{N} (i_{\mathsf{int}}\, q^i)\, s$$

  (where $q\, q^i\, s$ is a suitably relabelled position from $\mathsf{id}_{T([\![\theta]\!])}$, so that the moves involved are those from the ith copy of $T([\![\theta]\!])$ on the left and the copy on the right)
- $\mathsf{ref} : T([\![\mathsf{int}]\!]) \Rightarrow T([\![\mathsf{int\,ref}]\!])$

$$q\, q_{\mathsf{int}} \sum_{i=0}^{N} (i_{\mathsf{int}} \star (\mathit{read}\; i)^*)\, (\sum_{n=0}^{N} (\mathit{write}(n)\; \mathit{ok}\; (\mathit{read}\; n)^*))^*$$

- $\mathsf{assign} : T([\![\mathsf{int\,ref}]\!]) \times T([\![\mathsf{int}]\!]) \Rightarrow T([\![\mathsf{unit}]\!])$

$$r\, q_{\mathsf{int\,ref}}\ \star_{\mathsf{int\,ref}}\ q_{\mathsf{int}} \sum_{i=0}^{N} (i_{\mathsf{int}}\; \mathit{write}(i)_{\mathsf{int\,ref}})\; \mathit{ok}_{\mathsf{int\,ref}}\; d$$

- $\mathsf{deref} : T([\![\mathsf{int\,ref}]\!]) \Rightarrow T([\![\mathsf{int}]\!])$

$$q\, q_{\mathsf{int\,ref}}\ \star_{\mathsf{int\,ref}}\ \mathit{read}_{\mathsf{int\,ref}} \sum_{i=0}^{N} (i_{\mathsf{int\,ref}}\; i)$$

- $\mathsf{mkvar} : T([\![\mathsf{unit} \to \mathsf{int}]\!]) \times T([\![\mathsf{int} \to \mathsf{unit}]\!]) \Rightarrow T([\![\mathsf{int\,ref}]\!])$

$$q\, q^1\ \star^1\ q^2\ \star^2\ \star\ (\sum_{i=0}^{N} (\mathit{read}\; q^{1,\star}\; i^{1,\star}\; i\ +\ \mathit{write}(i)\; r^{2,i}\; d^{2,i}\; \mathit{ok}))^*$$

- $\mathsf{while} : T([\![\mathsf{int}]\!]) \times T([\![\mathsf{unit}]\!]) \Rightarrow T([\![\mathsf{unit}]\!])$

$$r\ (q_{\mathsf{int}}\ (\sum_{i=1}^{N} i_{\mathsf{int}})\; r_{\mathsf{unit}}\; d_{\mathsf{unit}})^*\; q_{\mathsf{int}}\; 0_{\mathsf{int}}\; d$$

Example 17 We revisit the terms $\mathsf{let}\, v = \mathsf{ref}(0)\, \mathsf{in}\, \lambda x.((v := x + !v); !v)$ and $\lambda x.\mathsf{let}\, v = \mathsf{ref}(0)\, \mathsf{in}\, ((v := x + !v); !v)$ from Example 3. Both are interpreted by a singleton family of strategies on $T([\![\mathsf{int} \to \mathsf{int}]\!]) = \sum_\star !(\prod_{i=0}^{N} T([\![\mathsf{int}]\!]))$. $q \star q^1\, 1^1\, q^3\, 4^3\, q^1\, 5^1$ is generated by the first term. Note that v is shared by all !-threads. The strategy interpreting the second term has the position $q \star q^1\, 1^1\, q^3\, 3^3\, q^1\, 1^1$ instead. Here each !-thread has its own copy of v.

The game model described in this section is fully abstract for $\precsim$ and $\cong$.

Definition 18 A play is *complete* if all questions therein have been answered.

Let $\mathsf{comp}(\sigma)$ denote the set of non-empty complete plays of a strategy. Then we have

Theorem 19 (Full Abstraction [11,19]) *Let* $\Gamma \vdash M_1, M_2 : \theta$, $[\![\Gamma \vdash M_1]\!] = \{\sigma_i\}_{i\in I}$, $[\![\Gamma \vdash M_2]\!] = \{\tau_i\}_{i\in I}$. *Then* $M_1 \precsim M_2 : \theta$ *if and only if for all* $i \in I$, $\mathsf{comp}(\sigma_i) \subseteq \mathsf{comp}(\tau_i)$. *Consequently, we have* $\Gamma \vdash M_1 \cong M_2$ *iff* $\mathsf{comp}(\sigma_i) = \mathsf{comp}(\tau_i)$ *for all* $i \in I$.

Unlike in the call-by-name case, complete plays are no longer maximal. This allows the game model to distinguish, for example, $\Omega_{\mathsf{int}\to\mathsf{unit}}$ from $\lambda x^{\mathsf{int}}.\Omega_{\mathsf{unit}}$: the former does not produce any non-empty complete position while the latter defines the position $q\,\star$.

4 When are pointers really needed?

Since we are going to represent positions using sequences of moves, it is vital to understand when this indeed leads to a faithful representation of positions, i.e. when the justification pointers can be uniquely reconstructed. The need for pointers in call-by-name game semantics arises at third order, where they are needed to distinguish the semantics of $\lambda f.f(\lambda x_0.f(\lambda x_1.x_0))$ from that of $\lambda f.f(\lambda x_0.f(\lambda x_1.x_1))$ (both terms of type $((\mathsf{unit}\to\mathsf{unit})\to\mathsf{unit})\to\mathsf{unit}$) [14]. Up to second-order in the call-by-name case they are completely redundant [2]. The call-by-value case is quite different. Because in call-by-value games answers can justify questions, new shapes of arenas arise and sometimes pointers have to be represented already at first order. The fact that ambiguities arise when pointers are removed stems from the use of $!G$-games. In $!G$, new threads of G can be started, but when occurrences of ! are nested, it becomes necessary to assign the inner threads to the outer ones and this is what pointers are used for. In call-by-value games the $!G$ games arise either through the function space construction or the strong monad T. In what follows we aim to identify terms and types whose game semantics does not need pointers. Because of the bracketing condition, it is not necessary to have pointers for answers, but there are many arenas where the same cannot be said about questions.

The enabling relation of an arena generated by an $\mathsf{RML_f}$ type can be thought of as a forest. Since the ability to retrieve pointers depends only on that relation,

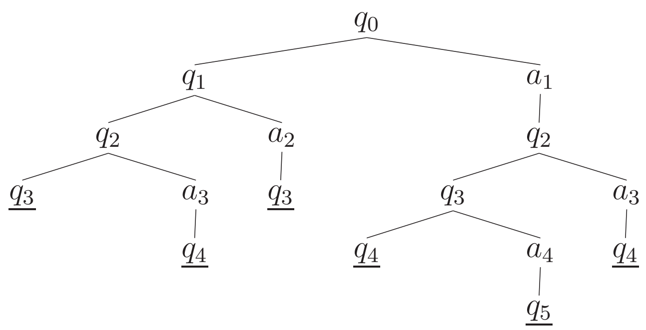

Fig. 4. Possible branches in the enabling relation

it suffices to analyze all possible shapes of branches that might arise. They are shown in Figure 4 in a compact form: the branches starting at the root and ending with a node that is *not* underlined turn out to be "safe": whenever they occur in an arena justification pointers from moves belonging to them can be omitted (i.e. uniquely reconstructed). This is not true for branches with underlined nodes, where forgetting the pointer from the underlined move leads to ambiguities as we show below, enumerating the six underlined nodes from left to right. In each row we give two positions that are different only thanks to the pointers, when pointers are removed they become identical. The branch $qqq\underline{q}$ (Case 1.) corresponds to the call-by-name example which we gave at the beginning of this section.

1.	$q_0\ q_1\ q_2\ q_1\ q_2\ q_3$	$q_0\ q_1\ q_2\ q_1\ q_2\ q_3$
2.	$q_0\ q_1\ q_2\ a_3\ q_2\ a_3\ q_4$	$q_0\ q_1\ q_2\ a_3\ q_2\ a_3\ q_4$
3.	$q_0\ q_1\ a_2\ q_1\ a_2\ q_3$	$q_0\ q_1\ a_2\ q_1\ a_2\ q_3$
4.	$q_0\ a_1\ q_2\ q_3\ q_2\ q_3\ q_4$	$q_0\ a_1\ q_2\ q_3\ q_2\ q_3\ q_4$
5.	$q_0\ a_1\ q_2\ q_3\ a_4\ q_3\ a_4\ q_5$	$q_0\ a_1\ q_2\ q_3\ a_4\ q_3\ a_4\ q_5$
6.	$q_0\ a_1\ q_2\ a_3\ q_2\ a_3\ q_4$	$q_0\ a_1\ q_2\ a_3\ q_2\ a_3\ q_4$

Theorem 20 *Pointers leading from questions in safe branches can be reconstructed uniquely (once erased).*

PROOF. We analyze all the five cases corresponding to the non-underlined question-nodes of the tree. The first case of an initial question is very easy, because initial moves never have pointers. So, four cases remain.

Recall that we consider well-opened games only, i.e. there is only ever one occurrence of the initial question. Hence there is no need to represent pointers to the initial question. In the remaining three cases (qqq, qaq, $qaqq$) the visibility condition will help: it turns out that only one of the potential justifiers is visible, so because the visibility condition must be satisfied, the pointer must point at the unique visible justifier.

($q_0 m_1 q_2$) Suppose sq_2 is a position. Note that if m_1 occurs in $\llcorner s \lrcorner$ then the preceding move must be q_0. Therefore there can only be one occurrence of m_1 in $\llcorner s \lrcorner$.

($q_0 a_1 q_2 q_3$) Suppose sq_3 is a position. Take $\ulcorner s \urcorner$. Note that if q_2 occurs in $\ulcorner s \urcorner$ the preceding move is a_1. Since a_1 answers the unique initial question, there can be only one a_1 in $\ulcorner s \urcorner$. Therefore there is a unique occurrence of q_2 in $\ulcorner s \urcorner$.

Given an arena we can use Figure 4 to see immediately whether positions on the arena can be stripped of pointers without loss of information: this is the case when the enabling relation does not generate the problematic branches with underlined nodes. Next we examine the implications of our findings for $\mathsf{RML_f}$-types. Our ultimate aim is to design a restricted type system $\mathsf{RML_f^-}$ where pointers can be omitted. The $\mathsf{RML_f^-}$ typing judgments will be those of $\mathsf{RML_f}$ subject to further type constraints:

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

In what follows we discuss what types should be allowed as *ctype* and *ttype*. Recall that the typing judgment above is interpreted by a family of strategies for the game

$$\eth = [\![\mathit{ctype}_1]\!] \times \cdots \times [\![\mathit{ctype}_n]\!] \Rightarrow T([\![\mathit{ttype}]\!]).$$

4.1 Curried functions

Problems with pointers turn out to arise whenever types of the shape $A \to (B \to C)$ are used (both as *ctype* and *ttype*). For example, suppose $\mathit{ctype} = \mathsf{unit} \to (\mathsf{unit} \to \mathsf{unit})$, which is actually the simplest such type. We have $[\![\mathit{ctype}]\!] = \{\sum_\star !T([\![\mathsf{unit}]\!])\}$, where the enabling relation, shown below, has the

$qaqa$-branch [3]

$$\begin{array}{ll} O & q \\ P & \star \\ O & r \\ P & d \end{array}$$

Then $\Game$ will have the problematic $qqaq$-branch and when pointers are left out, the positions generated by the terms M_0, M_1, defined by

$$M_i \equiv f : \mathsf{unit} \rightarrow (\mathsf{unit} \rightarrow \mathsf{unit}) \ \vdash \ \mathsf{let}\, g_0 = f()\,\mathsf{in}\,(\mathsf{let}\, g_1 = f()\,\mathsf{in}\, g_i()) : \mathsf{unit},$$

will be identified, though the terms generate the following different (unique) complete plays:

$$r_0 \quad q \quad \star \quad q \quad \star \quad r \quad d \quad d_0 \qquad\qquad r_0 \quad q \quad \star \quad q \quad \star \quad r \quad d \quad d_0$$

This will force us to exclude types of the form $A \rightarrow (B \rightarrow C)$ as *ctype*'s in the definition of $\mathsf{RML}_\mathsf{f}^-$. Similarly, whenever a closed term of type $ttype = \mathsf{unit} \rightarrow (\mathsf{unit} \rightarrow \mathsf{unit})$ is considered, $\Game = T([\![ttype]\!]) = \sum_\star !_1(\sum_\star !_2 T([\![\mathsf{unit}]\!]))$ will have the $qaqaq$-branch

$$\begin{array}{ll} O & q \\ P & \star \\ O & q_1 \\ P & \star_1 \\ O & r \\ P & d \end{array}$$

Pointers are then needed to disambiguate the following two positions

$$q \quad \star \quad q_1 \quad \star_1 \quad q_1 \quad \star_1 \quad r \quad d \qquad\qquad q \quad \star \quad q_1 \quad \star_1 \quad q_1 \quad \star_1 \quad r \quad d$$

Their origin is somewhat more complicated than in previous examples. They are induced respectively by N_0, N_1 defined by

$$\begin{aligned} N_i \equiv \mathsf{let}\ X = \mathsf{ref}\, 0\ \mathsf{in}\ (\lambda a.\ \mathsf{let}\ Y = \mathsf{ref}\, !X&, \\ \Box = ([!X \leq 1]; X := !X + 1)& \\ \mathsf{in}\ (\lambda b.\ [!X = 2; !Y = i]; Y := 2)&) \end{aligned}$$

---

[3] In order to simplify $[\![ctype]\!]$ we have used the fact that the game $1 \Rightarrow A$ is always isomorphic to A. Strictly speaking, $[\![ctype]\!] = \{1 \Rightarrow \sum_\star !(1 \Rightarrow T([\![\mathsf{unit}]\!]))\}$. We shall often do this in what follows.

To understand why this is the case, note that X is shared by all $!_1$-threads, while Y remains private to each $!_1$-thread and it is shared by all $!_2$-threads inside a fixed $!_1$-thread. The terms use the variables X and Y to restrict P's responses to only those required in the positions above. For instance, X is used to keep the total number of $!_1$ threads: each time one is opened, X is incremented. Then, because of $[!X \leq 1]$, P will reply to an O-move q_1 opening a $!_1$-thread only if this is the first or second such thread. Additionally, each $!_1$-thread has a private copy of Y which stores its index. Hence, because of $[!X = 2]$, attempts to open a $!_2$-thread (with r) can only trigger a reply from P if two $!_1$-threads have been opened before. Then the $!_2$-thread to be opened must be inside the $!_1$-thread numbered i.

The example shows that in general the game semantics of terms of type $\mathsf{unit} \to (\mathsf{unit} \to \mathsf{unit})$ cannot be described without pointers. However, banning all terms with types $A \to (B \to C)$ would deprive us of many interesting examples and prevent us from analyzing some aspects of RML references. Therefore, we are going to admit some terms with such types where the game semantics can be represented in a simpler way, by a (representative) sample of the full strategy. Needless to say, the two terms featured above cannot become part of $\mathsf{RML}_\mathsf{f}^-$.

The need for pointers in positions of $T([\![ttype]\!])$ is due to the fact that many $[\![ttype]\!]$-threads can be opened in the game $T([\![ttype]\!]) = \sum_\star ![\![ttype]\!]$ leading to multiple occurrences of q_1. The problem does not arise if $T'(\{A_i\}_{i\in I}) = \{\sum_{i\in I} A_i\}$ is used instead of the outer T (then we have $\sum_\star [\![ttype]\!]$). But when doing so we have to be sure that the restricted semantics still represents the full semantics faithfully. This will be the case when plays in each $[\![ttype]\!]$-thread are the same. Such plays are generated by λ-abstractions, free identifiers and preserved by case. Hence, we can admit terms of type $\mathsf{unit} \to (\mathsf{unit} \to \mathsf{unit})$ but only λ-abstractions possibly combined using case. We cannot introduce free identifiers though, because the previous example shows that their contraction could not be handled anyway. The restriction of T to T' means that we are representing single uses of the term only, therefore such terms cannot be used as arguments to functions, although they may be functions that are used in applications (application is a "linear" operation with respect to the function).

4.2 Higher-order types

The call-by-name example of third-order terms reappears for call-by-value at order four and shows that free variables of type $((\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \to \mathsf{unit}$ need pointers to be accounted for adequately. For $i = 0, 1$ the typing judgments

$$f : ((\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \to \mathsf{unit} \ \vdash \ f(\lambda x_1^{\mathsf{unit}\to\mathsf{unit}}.f(\lambda x_0^{\mathsf{unit}\to\mathsf{unit}}.x_i()))$$

generate the following positions respectively

$$r \quad r_1 \quad r_2 \quad r_1 \quad r_2 \quad r_3 \qquad\qquad r \quad r_1 \quad r_2 \quad r_1 \quad r_2 \quad r_3$$

Consequently, in $\mathsf{RML}_{\mathrm{f}}^{-}$ we cannot admit the above type as *ctype*. However, Figure 4 shows that there are no pointer-related problems for free identifiers of type $ctype = (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$, since the corresponding arena $[\![ctype]\!] \Rightarrow T([\![ctype]\!])$, shown below,

$$\begin{array}{llll}
\mathrm{O} & & & q_0 \\
\mathrm{P} & & r_1' & \star_0 \\
\mathrm{O} & r_2' & d_1' & r_1 \\
\mathrm{P} & d_2' & r_2 & d_1 \\
\mathrm{O} & & d_2 &
\end{array}$$

has branches of type qqq and $qaqq$. However, a new problem arises, because the semantics of such variables is not regular. $[\![f : ctype \vdash f : ctype]\!]$ contains positions of the following shape

$$q_0 \quad \star_0 \quad (r_1 \quad r_1' \quad r_2' \quad r_2)^n \quad (\,d_2 \quad d_2' \quad d_1' \quad d_1\,)^n$$

and cannot be captured as a regular language. The same obstacle occurs for other terms of type $(\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$, e.g. $\lambda f.f()$ has positions

$$q_0 \quad \star_0 \quad (\,r_1 \quad r_2\,)^n \quad (\,d_2 \quad d_1\,)^n$$

Like before, we will address the problem by replacing the full strategy on $\ldots \Rightarrow T([\![ttype]\!])$ with its restriction to $\ldots \Rightarrow T'([\![ttype]\!])$. As already remarked, this simplified representation is faithful for λ-abstractions and free identifiers (possibly combined using case).

To summarize, without pointers we can account for *ctype*'s of the form unit, $\mathsf{unit} \to \mathsf{unit}$, $(\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$ (but not $\mathsf{unit} \to (\mathsf{unit} \to \mathsf{unit})$). As for *ttype*, we can faithfully represent unit, $\mathsf{unit} \to \mathsf{unit}$ and use an impoverished, yet representative, version of the semantics for $\mathsf{unit} \to (\mathsf{unit} \to \mathsf{unit})$, $(\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$ and $(\mathsf{unit} \to \mathsf{unit}) \to (\mathsf{unit} \to \mathsf{unit})$. Roughly speaking, whenever the type of a term contains two occurrences of $\to$, we have to appeal to the simplified representation. Of course, the summary above is also relevant for types built from int (which is like unit as far as pointers are concerned) and $\mathsf{int\,ref}$ (which is like $\mathsf{unit} \to \mathsf{unit}$).

5 $\mathsf{RML}_\mathrm{f}^-$

We define a restriction $\mathsf{RML}_\mathrm{f}^-$ of RML_f, closely following the points made in the previous section. $\mathsf{RML}_\mathrm{f}^-$ will allow procedures with functional arguments and constructs $\mathsf{let}\, f = M \,\mathsf{in}\, N$ where both M and N are functions or references. For instance, terms discussed in Examples 3, 14, 16 will be $\mathsf{RML}_\mathrm{f}^-$ terms. We are going to prove that the game semantics of $\mathsf{RML}_\mathrm{f}^-$ programs can be represented by regular languages and, hence, the associated problems of equivalence and approximation are decidable. Initially, we shall focus on a type system without products but, since they are already present in the model, their addition is quite straightforward and at the same time useful, as we can then create tuples containing both functions and variables, i.e. a rough form of objects.

$\mathsf{RML}_\mathrm{f}^-$ is a subset of RML_f with typing judgments of the form

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

where *ctype* and *ttype* are as follows:

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

Example 21 (Memoisation [9]) The term $\mathit{memoise} : (\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})$ defined by

$$\begin{array}{l} \lambda f.\mathsf{let}\ a = \mathsf{ref}(0),\ r = \mathsf{ref}(f(0)) \\ \quad \mathsf{in}\ (\lambda x.(\mathsf{if}\ x = !a\ \mathsf{then}\ ()\ \mathsf{else}\ (a := x; r := f x)); !r) \end{array}$$

is in $\mathsf{RML}_\mathrm{f}^-$. $\mathit{memoise}\, f$ stores the value of the latest function call to f for future use thus improving efficiency. It is not true in general that $\mathit{memoise}\, f \cong f$, because f might use global store or depend on it. However, when f itself is an $\mathsf{RML}_\mathrm{f}^-$ term, it will be possible to use our framework to verify whether memoisation can be performed without observable effects.

Consider the game in which the judgments are interpreted in the game model, namely $[\![\mathit{ctype}_1]\!] \times \cdots \times [\![\mathit{ctype}_n]\!] \Rightarrow T([\![\mathit{ttype}]\!])$. Observe that branches contributed by $[\![\mathit{ctype}_i]\!]$ will always have safe shapes (Figure 4). Those originating from *ttype* are safe only if *ttype* is generated by α. As explained before, for *ttype*'s of the shape $\alpha \to \alpha$ (that are not α at the same time) we will resort to T' instead. Note that the only terms with such types are either λ-abstractions and free identifiers possibly in case-branches. Therefore, the full semantics can be recovered from the simplified one. In addition, the simplified version can be used for calculating the full semantics of subsequent applications of these terms: terms of type $\alpha \to \alpha$ (and not α) are later used only in applications or branching which can be interpreted faithfully with the "single use" representation.

Definition 22 Given an $\mathsf{RML}_{\mathsf{f}}^{-}$ judgment $[\![\Gamma \vdash M : \mathit{ttype}]\!]$ let $[\![\Gamma]\!] = \{A\}_{i\in I}$. Then $[\![\Gamma \vdash M : \mathit{ttype}]\!] = \{\sigma_i\}_{i\in I}$, where $\sigma_i : A \Rightarrow T([\![\mathit{ttype}]\!])$. We shall write $\|\Gamma \vdash M : \mathit{ttype}\|$ for $\{\rho_i\}_{i\in I}$ where

$$\rho_i = \begin{cases} \mathsf{comp}(\sigma_i) & \mathit{ttype} = \alpha \\ \mathsf{comp}(\sigma_i) \cap P_{A\Rightarrow T^{-}([\![\mathit{ttype}]\!])} & \mathit{ttype} \neq \alpha \end{cases}$$

Proposition 23 *For $\mathsf{RML}_{\mathsf{f}}^{-}$ terms $\Gamma \vdash M_1, M_2 : \mathit{ttype}$, let $\|\Gamma \vdash M_j\| = \{\rho_i^j\}_{i\in I}$. Then $\Gamma \vdash M_1 \sqsubseteq M_2$ if and only if $\rho_i^1 \subseteq \rho_i^2$ for all $i \in I$.*

Theorem 24 *For any $\mathsf{RML}_{\mathsf{f}}^{-}$ typing judgment $\Gamma \vdash M : \mathit{ttype}$, let $\|\Gamma \vdash M : \mathit{ttype}\| = \{\rho_i\}_{i\in I}$. Then ρ_i is a regular language over $M_{A\Rightarrow T([\![\mathit{ttype}]\!])} = M_A + M_{T([\![\mathit{ttype}]\!])}$ for all $i \in I$.*

PROOF. We follow the style of presentation from the tutorial article [22]. Let $[\![\Gamma]\!] = \{A\}_{i\in I}$, $[\![\alpha]\!] = \{B\}_{j\in J}$. We consider free identifiers first.

($\mathit{ctype} = \alpha$) Then $[\![\Gamma, \alpha]\!] = \{A \times B\}_{(i,j)\in I\times J}$ and consequently $\|\Gamma, x : \alpha \vdash x : \alpha\| = \{\rho_{i,j}\}_{(i,j)\in I\times J}$. Then $\rho_{i,j} = q\, j\, C_\alpha^*$, where

$$\begin{aligned} C_\beta &= \emptyset \\ C_{\mathsf{int\,ref}} &= \textstyle\sum_{n=0}^{N} (\mathit{write}(n)_r\ \mathit{write}(n)_l\ \mathit{ok}_l\ \mathit{ok}_r + \mathit{read}_r\ \mathit{read}_l\ n_l\ n_r) \\ C_{\beta_1\to\beta_2} &= \textstyle\sum_{u\in U} (q_r^u q_l^u \sum_{v\in V} (v_l^u\, v_r^u)) \end{aligned}$$

where $\beta_1 = \{1\}_{u\in U}$, $\beta_2 = \{1\}_{v\in V}$ (i.e. $[\![\beta_1 \to \beta_2]\!] = \prod_{u\in U} T([\![\beta_2]\!])$).

($\mathit{ctype} = \alpha \to \beta$) Then $[\![\mathit{ctype}]\!] = \{\prod_{j\in J}(B \Rightarrow T([\![\beta]\!]))\}$, so $[\![\Gamma, \mathit{ctype}]\!] = \{A \times [\![\mathit{ctype}]\!]\}_{(i,\star)\in I\times\{\star\}}$ and $[\![\Gamma, x : \mathit{ctype} \vdash x : \mathit{ctype}]\!] = \{\rho_{i,\star}\}_{(i,\star)\in I\times\{\star\}}$. Supposing $[\![\beta]\!] = \{1\}_{v\in V}$ we have $\rho_{i,\star} = q \star C_{\alpha\to\beta}$ where

$$C_{\alpha\to\beta} = \sum_{j\in J} (q_r^j q_l^j (C_\alpha[l, r/r, l])^* \sum_{v\in V} (v_l^j v_r^j)).$$

For λ-abstraction, suppose $\|\Gamma \vdash \lambda x^{\alpha_1}.M : \mathit{ttype}\| = \{\rho_i\}_{i\in I}$, $\mathit{ttype} = \alpha_1 \to \alpha_2$ and $\|\Gamma, x : \alpha_1 \vdash M : \alpha_2\| = \{\rho_i'\}_{i\in I}$. Then

$$\rho_i = \begin{cases} q \star (\rho_i')^* & \mathit{ttype} = \alpha \\ q \star \rho_i' & \mathit{ttype} \neq \alpha \end{cases}$$

Let $\{X_z\}_{z\in Z}$ be a family of sets and let $Y \subseteq Z$. Below we are going to use homomorphisms $\phi : (\sum_{z\in Z} X_z)^* \rightarrow (\sum_{y\in Y} X_y)^*$ such that for $x \in \sum_{z\in Z} X_z$:

$$\phi(x) = \begin{cases} x & x \in X_z,\ z \in Y \\ \epsilon & \text{otherwise} \end{cases}$$

i.e., depending on whether $z \in Y$, the character is copied or erased. We shall always use the letter ϕ to refer to homomorphisms of that kind. Observe that for $L \subseteq (\sum_{y\in Y} X_y)^*$, $\phi^{-1}(L)$ consists of words $l \in L$ which are padded with characters from X_z ($z \in Z \setminus Y$). Recall that homomorphic images and inverse images preserve regularity [23].

Next we consider constructs interpreted by the special strategies ref, assign, deref, while, mkvar (case and apply will be treated separately in a more direct way in order to avoid characterizing them for each θ in turn).

Let $\sigma_{\mathsf{op}} : T(\llbracket ttype_1 \rrbracket) \times T(\llbracket ttype_2 \rrbracket) \Rightarrow T(\llbracket ttype \rrbracket)$ be the strategy used to interpret an $\mathsf{RML}_{\mathsf{f}}^{-}$ typing judgment of the shape

$$\frac{\Gamma \vdash M_1 : ttype_1 \quad \Gamma \vdash M_2 : ttype_2}{\Gamma \vdash \mathsf{op}(M_1, M_2) : ttype}.$$

Subsection 3.4 shows that $\mathsf{comp}(\sigma_{\mathsf{op}})$ is regular. Suppose $\|\Gamma \vdash M_j : ttype_j\| = \{\rho_i^j\}_{i\in I}$ for $j = 1, 2$, let $\mathcal{A} = M_A + M_{T(\llbracket ttype_1 \rrbracket)} + M_{T(\llbracket ttype_2 \rrbracket)} + M_{T(\llbracket ttype \rrbracket)}$ and

$$\begin{aligned} \phi &: \mathcal{A}^* \rightarrow (M_{T(\llbracket ttype_1 \rrbracket)} + M_{T(\llbracket ttype_2 \rrbracket)} + M_{T(\llbracket ttype \rrbracket)})^* \\ \phi_{1,2} &: \mathcal{A}^* \rightarrow (M_A + M_{T(\llbracket ttype_1 \rrbracket)} + M_{T(\llbracket ttype_2 \rrbracket)})^* \\ \phi' &: \mathcal{A}^* \rightarrow (M_A + M_{T(\llbracket ttype \rrbracket)})^* \end{aligned}$$

Then $\|\Gamma \vdash \mathsf{op}(M_1, M_2)\| = \{\rho_i\}_{i\in I}$ where

$$\rho_i = \phi'(\quad \phi_{1,2}^{-1}(\quad \rho_i^1 \,||\, \rho_i^2 \quad) \cap \phi^{-1}(\mathsf{comp}(\sigma_{\mathsf{op}})\ )).$$

The above formula simply describes the interleaving of positions controlled by σ_{op}. For $\mathsf{op} = \mathsf{while}$, one has to use $(\rho_i^1)^*$ and $(\rho_i^2)^*$ instead of ρ_i^1, ρ_i^2.

Finally we consider branching and application.

(case_{ttype}) Suppose $\|\Gamma \vdash M : \mathsf{int}\| = \{\rho_i'\}_{i\in I}$, $\|\Gamma \vdash M_n : ttype\| = \{\rho_i^n\}_{i\in I}$ for $n = 0, \cdots, N$, and $\|\Gamma \vdash \mathsf{case}_{ttype}(M)[M_0, \cdots, M_N]\| = \{\rho_i\}_{i\in I}$. Observe that, roughly speaking, each position in ρ_i arises from a position of M_n preceded by a position from ρ_i' that ends in n. We capture that intuition formally next. For each $i \in I$, we decompose ρ_i' into $\bigcup_{n=0} \rho_{in}'$ where positions of ρ_{in}' are those ending in n. ρ_{in}' are still regular languages, because $\rho_{in}' = \rho_i' \cap (q_{\mathsf{int}}\, M_A^*\, n_{\mathsf{int}})$. In what follows we shall interleave plays of ρ_{in}' with those of ρ_i^n in the same way as this happens during composition with case_θ. Let $\mathcal{A} = M_A + M_{T([\![\mathsf{int}]\!])} + M_{T([\![ttype]\!])}$ and $\mathcal{A}_1 = M_A + M_{T([\![ttype]\!])}$. Then

$$\rho_i = \phi(\quad \sum_{n=0}^{N} (\rho'_{in} || \rho_i^n) \ \cap \ q\, q_{\mathsf{int}}\, M_A^* \, (\sum_{n=0}^{N} n_{\mathsf{int}}) \, (\mathcal{A}_1 \setminus \{q\})^* \quad)$$

where $\phi : \mathcal{A}^* \to \mathcal{A}_1^*$.

(apply) Suppose $\|\Gamma \ \vdash \ M_1 : \alpha \to \alpha'\| = \{\rho_i^1\}_{i\in I}$, $\|\Gamma \ \vdash \ M_2 : \alpha\| = \{\rho_i^2\}_{i\in I}$ and $\|\Gamma \ \vdash \ M_1 M_2 : \alpha'\| = \{\rho_i\}_{i\in I}$. Then ρ_i^1 and ρ_i^2 are regular languages for the alphabets $\mathcal{A}^1, \mathcal{A}^2$ respectively:

$$\begin{aligned}
\mathcal{A}^1 &= M_A + M_{T([\![\alpha]\!] \Rightarrow T([\![\alpha']\!]))} = M_A + \{q^1, \star\} + \textstyle\sum_{j\in J} M_{B \Rightarrow T([\![\alpha']\!])} = \\
&= M_A + \{q^1, \star\} + \textstyle\sum_{j\in J} M_B + \sum_{j\in J} T([\![\alpha']\!]) \\
\mathcal{A}^2 &= M_A + M_{T([\![\alpha]\!])} = M_A + \{q^2\} + J + \textstyle\sum_{j\in J} M_B.
\end{aligned}$$

Note that $\mathcal{A}^1$ and $\mathcal{A}^2$ share $\sum_{j\in J} M_B$. The composition of ρ_i^1 and ρ_i^2 with $\mathsf{apply}_{[\![\alpha]\!],[\![\alpha']\!]}$ can be thought of as synchronization on that component. That interaction can be captured by words over

$$\mathcal{A} = M_A + \overbrace{\textstyle\sum_{j\in J} T([\![\alpha']\!]) + \{q^1, \star\} + \underbrace{\textstyle\sum_{j\in J} M_B}}^{\mathcal{A}^1}\underbrace{{} + \{q^2\} + J + M_A}_{\mathcal{A}^2}$$

Additionally, Stage 1 of ρ_i^2 has to be scheduled right after $\star$. We will model that by additional synchronization on q^2 and elements of J which are inserted into ρ_i^1 following $\star$:

$$\rho_i^{1'} = (\ \rho_i^1 \,||\, (q^2\, J)\ ) \cap (\ q^1 \mathcal{A}^* \star q^2\, J\, \mathcal{A}^*\ ).$$

Subsequently the synchronizing moves will be erased using γ. Let $\mathcal{A}_1 = \mathcal{A}^1 + \{q^2\} + J$, $\mathcal{A}_2 = \mathcal{A}^2$, and let $\phi_i : \mathcal{A}^* \to \mathcal{A}_i^*$ for $i = 1, 2$. Let $\gamma : \mathcal{A}^* \to (M_A + M_{T([\![\alpha']\!])})^\star$ be the homomorphism that copies elements of the two copies of M_A and those from $\sum_{j\in J} M_{T([\![\alpha']\!])}$ to their unique copies on the right, and erases $q^1, \star, q^2$, elements of J and $\sum_{j\in J} M_B$. Then $\rho_i = \gamma(\phi_1^{-1}(\rho_i^{1'}) \cap \phi_2^{-1}(\rho_i^2))$.

5.1 Addition of products

We extend the previously given type system by considering typing judgments of the form $x_1 : ctype_1, \cdots, x_n : ctype_n \ \vdash \ M : ttype$, where

$$\begin{aligned}
ctype &::= \alpha' \mid \alpha' \to \beta' & \qquad ttype &::= \alpha' \mid \alpha' \to \alpha' \\
\alpha' &::= \alpha \times \cdots \times \alpha & \beta' &::= \beta \times \cdots \times \beta \\
\alpha &::= \beta \quad \mid \quad \beta' \to \beta' \quad \mid \quad \mathsf{int\,ref} & \beta &::= \mathsf{unit} \quad \mid \quad \mathsf{int}
\end{aligned}$$

We will still use the name $\mathsf{RML}_{\mathsf{f}}^{-}$ for the resulting language. $\mathsf{RML}_{\mathsf{f}}^{-}$ supports tupling of terms of types α (i.e. variables, first-order functions and ground-type values) through the following rules

$$\frac{\Gamma \vdash M_i : \alpha_i \qquad i = 1, \cdots, k}{\Gamma \vdash \langle M_1, \cdots, M_k \rangle : \alpha_1 \times \cdots \times \alpha_k} \quad \frac{\Gamma \vdash M : \alpha_1 \times \cdots \times \alpha_k \qquad 1 \leq i \leq k}{\Gamma \vdash \pi_i(M) : \alpha_i}$$

with the following operational semantics

$$\frac{M_1 \Downarrow V_1 \quad \cdots \quad M_k \Downarrow V_k}{\langle M_1, \cdots, M_k \rangle \Downarrow \langle V_1, \cdots, V_k \rangle} \qquad \frac{M \Downarrow \langle V_1, \cdots, V_k \rangle}{\pi_i(M) \Downarrow V_i}.$$

$\mathsf{let}\,\langle x_1, \cdots, x_k \rangle = M_1\ \mathsf{in}\ M_2$ will serve as shorthand for

$$\mathsf{let}\, x = M_1,\ x_1 = \pi_1(x),\ \cdots,\ x_k = \pi_k(x)\ \mathsf{in}\ M_2.$$

Example 25 (Equality test for locations) The equality test for locations $eq : \mathsf{int\,ref} \times \mathsf{int\,ref} \to \mathsf{int}$ can be defined in $\mathsf{RML}_{\mathsf{f}}^{-}$ by

$$\lambda(x, x').\mathsf{let}\,\langle v, b \rangle = \langle\, !x,\ \ x := !x' + 1; (!x = !x')\,\rangle\ \mathsf{in}\ (x := v; b).$$

See also Examples 27 and 28.

On the semantic front, since $\mathit{Fam}(\mathcal{I})$ has products, we can again appeal to the standard interpretation [20]: projections are interpreted by composition with T-images of projections from $\mathit{Fam}(\mathcal{I})$ (which we simply call proj^i), for tupling one uses the previously discussed left-maps:

$$\begin{array}{rl} \mathsf{proj}^i_{\alpha_1,\cdots,\alpha_k} & : T(\llbracket \alpha_1 \rrbracket \times \cdots \times \llbracket \alpha_k \rrbracket) \Rightarrow T(\llbracket \alpha_i \rrbracket) \\ \mathsf{left}_{\alpha_1,\cdots,\alpha_k} & : T(\llbracket \alpha_1 \rrbracket) \times \cdots \times T(\llbracket \alpha_k \rrbracket) \Rightarrow T(\llbracket \alpha_1 \times \cdots \times \alpha_k \rrbracket). \end{array}$$

Assuming $\llbracket \alpha_i \rrbracket = \{A_i\}_{j_i \in I_i}$ for $i = 1, \cdots, k$, the complete positions of tuple and proj are of the following shapes:

$$\begin{array}{rl} \mathsf{comp}(\mathsf{proj}^i_{\alpha_1,\cdots,\alpha_k}) & = q\, q^1\, (j^1_1, \cdots, j^1_k)\, j_i\, C^*_{\alpha_i} \\ \mathsf{comp}(\mathsf{left}_{\alpha_1,\cdots,\alpha_k}) & = q\, q^1\, j^1_1\, \cdots\, q^k\, j^k_k\, (j_1, \cdots, j_k)\, (\sum_{i=1}^{k} C_{\alpha_i})^* \end{array}$$

where C_α is the same as in the proof of Theorem 24. Observe that the new types do not cause any new problems with pointers, since the product game construction does not contribute any new branches in the enabling relation. Therefore, like before, we will represent the full semantics if *ttype* is an α'-type and its simplified version (based on T^-) when *ttype* is not an α'-type. After changing α to α' we can adopt Definition 22 and prove that Theorem 24 still holds.

PROOF. Only the case of free identifiers needs to be revisited. For other constructs the same analysis still applies. Tupling and projecting can be interpreted following the pattern for σ_{op}.

Suppose $[\![\Gamma]\!] = \{A\}_{i\in I}$, $\alpha' = \alpha_1 \times \cdots \times \alpha_k$ and $[\![\alpha']\!] = \{B\}_{j\in J}$.

- $\mathit{ctype} = \alpha'$. Hence, $[\![\Gamma \times \mathit{ctype}]\!] = \{A \times B\}_{(i,j)\in I\times J}$, i.e. $\|\Gamma, x : \mathit{ctype} \vdash x : \mathit{ctype}\| = \{\rho_{(i,j)}\}_{(i,j)\in I\times J}$. Then we have $\rho_{(i,j)} = q\, j\, (C_{\alpha'})^*$, where $C_{\alpha'} = C_{\alpha_1} + \ldots + C_{\alpha_k}$, C_β, $C_{\mathsf{int\,ref}}$ are defined as in the proof of Theorem 24, and supposing[4] $[\![\beta_1']\!] = \{1^?\}_{u\in U}$ and $[\![\beta_2']\!] = \{1^?\}_{v\in V}$, (i.e. $[\![\beta_1' \to \beta_2']\!] = \prod_{u\in U} T([\![\beta_2']\!])$)

$$C_{\beta_1'\to\beta_2'} = \sum_{u\in U} (q_r^u\, q_l^u \sum_{v\in V} (v_l^u\, v_r^u)).$$

- $\mathit{ctype} = \alpha' \to \beta'$. Hence, $[\![\mathit{ctype}]\!] = \{\prod_{j\in J}(B \Rightarrow T([\![\beta']\!]))\}$ and $\|\Gamma, x : \mathit{ctype} \vdash x : \mathit{ctype}\| = \{\rho_{i,\star}\}_{(i,\star)\in I\times\{\star\}}$. Then we have $\rho_{i,\star} = q \star C_{\alpha'\to\beta'}$ where, supposing $\{\beta'\} = \{1^?\}_{v\in V}$, we set

$$C_{\alpha'\to\beta'} = \sum_{j\in J} (q_r^j\, q_l^j\, (C_{\alpha'}[r, l/l, r])^* \sum_{v\in V} (v_l^j\, v_r^j)).$$

The proof of Theorem 24 shows that the regular language representation can be generated from the term in an effective way. Hence we have

Theorem 26 *Observational equivalence and approximation of* $\mathsf{RML}_{\mathsf{f}}^{-}$*-terms are decidable.*

We finish this section with some equivalences previously considered in the literature. Because the terms involved turn out to belong to $\mathsf{RML}_{\mathsf{f}}^{-}$ they can be confirmed using Theorem 26.

Example 27 (Representation independence) The following equivalence between terms of type $(\mathsf{unit} \to \mathsf{unit}) \times (\mathsf{unit} \to \mathsf{int})$ is true:

$$\begin{array}{l}\mathsf{let}\ \langle\, c_1, c_2 \,\rangle = \langle\, \mathsf{ref}(0),\ \mathsf{ref}(1) \,\rangle \\ \mathsf{in}\ \langle\, \lambda x.(c_1 := !c_2; c_2 := !c_2 + 1),\ \ \lambda x.(!c_1 = !c_2) \,\rangle \ \cong\ \langle\, \lambda x.(),\ \ \lambda x.0 \,\rangle.\end{array}$$

Example 28 (Profiling[9]) Consider the term

$$\mathit{profile} \equiv \lambda f.\mathsf{let}\, c = \mathsf{ref}(0)\,\mathsf{in}\,\langle\, \lambda x.(c := !c + 1; f x),\ \lambda x.!c \,\rangle$$

of type $\alpha \to (\alpha \times (\mathsf{unit} \to \mathsf{int}))$, where $\alpha = \beta_1' \to \beta_2'$. For any $f : \alpha$, $\mathit{profile}\, f$ returns a tuple whose first component behaves exactly as f but in addition

[4] We write $1^?$ for $1 \times \cdots \times 1$.

each call increments c. The current total is then available via the second component. Both properties are formally captured by the equivalences: $f:\alpha \vdash \pi_1(\mathit{profile}\ f) \cong f$ and

$$\begin{array}{l} f:\alpha, \\ g,h:\alpha\rightarrow\mathsf{unit} \end{array} \vdash \begin{array}{l} \mathsf{let}\,\langle\, f', r\,\rangle = \mathit{profile}\ f \\ \mathsf{in}\ (gf'; fx; hf'; (r()+1)) \end{array} \cong \begin{array}{l} \mathsf{let}\,\langle\, f', r\,\rangle = \mathit{profile}\ f \\ \mathsf{in}\ (gf'; f'x; hf'; r()). \end{array}$$

Finally, we consider Example 5.9 from [9] which goes beyond $\mathsf{RML}_\mathsf{f}^-$:

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

The underlying game semantics is not regular in this case (it is deterministic context-free). A simplified version of this equivalence:

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

still fits into RML_f.

6 Undecidability

In this section we show that the halting problem for a class of finite-state machines equipped with queues can be reduced to a second-order program equivalence query. The discovery of universality of such machines goes back to Post's research on simple rewriting systems with undecidable termination problems [24]. Among the best known are the *normal* canonical systems wherein each reduction has the shape $a_i\Box \rightarrow \Box b_i$. With queues, rules like these are easily implementable and machines inspired by this observation were indeed considered by Post in the form of P-tag systems [25].

We are going to demonstrate that there are strategies induced by second-order RML_f terms which can be viewed as computations with queues. Informally, the queue structure will arise from suitably interleaved and nested function calls whose interaction is controlled by local variables with suitable scopes. In order to capture this behaviour we consider an auxiliary kind of store called a *Q-store*. A Q-store is an unbounded array where each entry, in addition to the stored element, contains two other fields used for bookkeeping. Any stored element can be accessed provided that some requirements, formulated using the bookkeeping information, are satisfied. The purpose of the bookkeeping is to make it possible to detect and isolate the case when the history of a Q-store corresponds to the queue discipline. Q-stores were first introduced in [17] where the undecidability of Idealized Algol was proved. Here we show how to adapt the approach to second-order RML_f.

In what follows we use a finite alphabet Σ on the understanding that it can be easily modelled using the ground datatype `int` since the latter contains at least two elements ($N > 0$).

Definition 29 A *Q-store* stores characters from Σ. Its content is defined by a natural number n (size) and a function $f : \{0, \cdots, n\} \to \Sigma \times \{+, -\} \times \{+, -\}$. The three fields of $f(i)$ will be referred to as $f(i).SYMBOL$, $f(i).ACCESSED$ and $f(i).MARKED$ respectively (the first one holds the stored character, the latter two indicate whether the ith entry has already been accessed or marked by the FETCH operation to be introduced later). The empty Q-store is defined by $n = 0$ and $f(0) = (\dagger, +, -)$, where $\dagger$ is a dummy symbol already deemed *accessed* but *unmarked*:

	0
SYMBOL	$\dagger$
ACCESSED	$+$
MARKED	$-$

A Q-store can be modified using only two operations:

- ADD x adds $x \in \Sigma$ to the store. The new Q-store

  $$f' : \{0, \cdots, n+1\} \to \Sigma \times \{+, -\}^2$$

  is defined by $f \subseteq f'$, $f'(n+1) = (x, -, -)$.
- FETCH is the only access method. It can return any previously unaccessed stored character $f(i).SYMBOL$ (i.e. $f(i).ACCESSED = -$) provided an index j can be found such that $0 \le j < i \le n$ and

  $$f(j).ACCESSED = +,\ f(j).MARKED = -.$$

  As a result, $f(i).ACCESSED, f(j).MARKED$ both change to $+$.

In short, FETCH can access an unaccessed element with index i provided there exists another unmarked but already accessed element with an index smaller than j. Note that the choice of (i, j) is an integral part of the access procedure and different choices affect the store in different ways.

Example 30 The Q-store shown below resulted from the following sequence of operations performed on the empty Q-store: ADD a, ADD b, FETCH, ADD c, ADD d, FETCH, ADD e ($(1, 0)$ and respectively $(4, 1)$ were chosen by FETCH). The accessed symbols were a and d.

	0	1	2	3	4	5
SYMBOL	†	a	b	c	d	e
ACCESSED	+	+	−	−	+	−
MARKED	+	+	−	−	−	−

Q-stores are nondeterministic. In particular the GET-FETCH behaviour might turn out to imitate a queue. This will occur when each FETCH operation chooses i and j to be the index of the first unaccessed element and $i-1$ respectively. The "first-in first-out" discipline leaves a characteristic pattern in the store: no minus occurs between two pluses in the row of *ACCESSED* fields, as shown in the store below (which arises when the two instances of FETCH from Example 30 choose $(1,0)$ and $(2,1)$).

	0	1	2	3	4	5
SYMBOL	†	a	b	c	d	e
ACCESSED	+	+	+	−	−	−
MARKED	+	+	−	−	−	−

Note that after an occurrence of $-$ arises between two pluses (like in Example 30) it can never be overwritten, because FETCH will be unable to find a suitable element for marking. Therefore, given a Q-store we can immediately tell whether it acted like a queue. Another consequence is the following

Lemma 31 *If all elements of a Q-store have been accessed, then its behaviour pattern was that of a queue.*

We introduce state machines equipped with Q-stores next.

Definition 32 A Q-machine is a tuple $\hat{Q} = \langle\, Q, \Sigma, q_0, F, \delta^{\mathrm{ADD}}, \delta^{\mathrm{FETCH}}\,\rangle$, where:

- $Q = Q^{\mathrm{A}} + Q^{\mathrm{F}} + F$ is the set of states, $q_0 \in Q^{\mathrm{A}}$;
- $\delta^{\mathrm{ADD}} : Q^{\mathrm{A}} \to Q \times \Sigma$ defines transitions from states $q \in Q^{\mathrm{A}}$: ADD $\pi_2(\delta^{\mathrm{ADD}}(q))$ is performed on the machine's Q-store and the state changes to $\pi_1(\delta^{\mathrm{ADD}}(q))$;
- $\delta^{\mathrm{FETCH}} : Q^{\mathrm{F}} \times \Sigma \to Q$ defines actions at states in Q^{F}: an attempt to execute FETCH takes place and if it is successful the next state depends on the returned symbol.

A Q-machine starts from the initial state q_0 with empty Q-store and makes (nondeterministic) transitions until a final state (from F) is reached. A run ending in a final state will be called *final*. A final run leading to a Q-store in which all elements have been accessed is called *complete*. We will say that a Q-machine *halts* if there exists a complete run. By the above Lemma, as far as halting is concerned, Q-machines are embellished queue-equipped finite-state automata. Hence, they essentially inherit the following Theorem from Post's work.

Theorem 33 *Q-machines have an undecidable halting problem.*

6.2 Representing Q-machines

We are going to define a representation scheme for arbitrary Q-machine runs. They will be represented via plays for a certain game, but eventually we will isolate only those positions that correspond to complete runs. For technical convenience we will assume that the initial state of a Q-store results from a special ADD action, indexed by 0, executed once at the very beginning, which introduces the unmarked accessed symbol † but does not affect the state.

We shall use plays for the game $\Game = [\![(\mathsf{unit} \to \mathsf{unit}) \to (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}]\!]$, i.e. $\sum_\star !(T([\![\mathsf{unit}]\!]) \Rightarrow \sum_\star !(T([\![\mathsf{unit}]\!]) \Rightarrow T([\![\mathsf{unit}]\!])))$. Its enabling relation is presented below:

$$
\begin{array}{lccc}
\mathrm{O} & & & q_0 \\
\mathrm{P} & & & \star_0 \\
\mathrm{O} & & & q \\
\mathrm{P} & r_1 & & \star \\
\mathrm{O} & d_1 & & r \\
\mathrm{P} & r_2 & & d \\
\mathrm{O} & d_2 & &
\end{array}
$$

r_1, d_1 originate from the left copy of $T([\![\mathsf{unit}]\!])$, r_2, d_2 come from the middle one and r, d from the right one.

The representation of a Q-machine run will begin with $q_0 \star_0$.

Each ADD operation (including the dummy operation initializing the store) will then be interpreted by the segment $q \star r\ r_2$, where the justification pointer of q points at the second "initializing" move $\star_0$:

$$q_0 \star_0 \quad \cdots \quad \underbrace{q \;\star\; r \;\; r_2}_{\mathrm{ADD}}$$

FETCH steps using (i, j) $(j < i)$ will be represented by segments $r r_1 r\ d d_1 d$, where the justification pointers of r, r_1 depend i and j as shown below:

$$q_0\ \star_0\ \cdots\ \underbrace{q\ \ \star\ \ r\ \ r_2}_{j\text{thADD}}\ \cdots\ \underbrace{q\ \ \star\ \ r\ \ r_2}_{i\text{thADD}}\ \cdots\cdots\ \underbrace{r\ r_1\ r\ d\ d_1\ d}_{\text{FETCH}}$$

Lemma 34 *The sequences of moves interpreting Q-store histories according to the above recipe are plays on* $\reflectbox{$\partial$}$.

PROOF. It suffices to verify the bracketing and visibility conditions. The former is easily seen to be satisfied so we focus on visibility.

Observe that the O-view of each such sequence consists of $q_0\star_0$, all ADD-blocks and segments rd originating from each FETCH-block.

(ADD) Because $q_0\star_0$ is visible, the first move of each ADD-block satisfies visibility. So do the following three moves since each of them is justified by the preceding move.

(FETCH) Because each ADD-block is visible, the first move r in a FETCH-block satisfies visibility. After r is played, the question q from the ith block is visible to P so the second move r_1 in the block is also correct. After the two moves, O still has all ADD blocks preceding the ith ADD block in his view, so the third move r in the block satisfies the rules. Finally, the three answers at the end of a FETCH block are easily seen to be correct. □

The link between game semantics and the halting problem for Q-machine will be provided by the following theorem. Strictly speaking, $[\![M_{\hat{Q}}]\!]$ is a singleton family of strategies, but we will simply treat it as a strategy.

Theorem 35 *For each Q-machine $\hat{Q}$ there exists a term $M_{\hat{Q}}$ such that*

$$\mathit{comp}([\![M_{\hat{Q}}]\!]) = \{q_0\ \star_0,\ q_0\ \star_0\ q\ \star\}$$

if and only if $\hat{Q}$ does not halt.

More precisely, we shall show

Lemma 36 *For a given Q-machine $\hat{Q}$, there exists a term $M_{\hat{Q}}$ such that $\mathit{comp}([\![M_{\hat{Q}}]\!])$ contains $q_0\ \star_0$, $q_0\ \star_0\ q\ \star$ as well as positions of the shape $s\,\underbrace{d_2d\cdots d_2d}_{n+1}$ where*

- *s is a position representing a complete run of $\hat{Q}$ resulting in a Q-store of size n;*
- *the $n+1$ d_2d-segments are answers to respectively r, r_2 played in ADD blocks numbered from n down to 0 (in fact there is no choice by the bracketing condition).*

Clearly, the above Lemma implies the Theorem: if $\hat{Q}$ never halts we have $\mathsf{comp}(\llbracket M_{\hat{Q}} \rrbracket) = \{q_0\star_0, q_0\star_0 q\star\}$; if it does then there will be another position in $\mathsf{comp}(\llbracket M_{\hat{Q}} \rrbracket)$ representing the halting (complete) run. Before $M_{\hat{Q}}$ is introduced we examine more closely a number of terms which will be used as its building blocks.

Let us consider the position $q_0\star_0 q\star$ which is generated, for example, by terms of the shape $\lambda f.\lambda g.(\cdots)$. Suppose O plays r justified by $\star$ next. Various replies are possible.

Example 37 (1) $\llbracket \lambda f.\lambda g.g() \rrbracket$ replies with r_2 producing $q_0\ \star_0\ q\ \star\ r\ \ r_2$, which is consistent with the way we interpret ADD operations. Moreover, when O subsequently plays d_2, P will reply with d (just like Lemma 36 expects the players to behave after a complete run). The play can go on (O could play new copies of q, r or d, rules permitting) but P's responses will always be analogous.

(2) $\llbracket \lambda f.\lambda g.f() \rrbracket$ replies with r_1 yielding $q_0\ \star_0\ q\ \star\ r\ \ r_1$. This time the responses match the first two moves used to represent FETCH. Moreover, when d_1 is played next, P will reply with d, as in the final moves of the representation of a FETCH step.

(3) $\llbracket \lambda f.\lambda g.() \rrbracket$ contains $q_0\ \star_0\ q\ \star\ r\ \ d$, which is the same as the middle part of the simulation of FETCH.

$M_{\hat{Q}}$ will imitate one of the above terms depending on the stage of the simulation. Recall that $\Game$ has the shape $\sum_\star(!_1(A \Rightarrow \sum_\star !_2(B \Rightarrow C)))$. ADD steps can then be viewed as opening new $!_1$-threads (with q) while FETCH steps revisit the two $!_1$-threads corresponding to i and j (with r). As for $!_2$-threads, each ADD step creates one $!_2$-thread nested inside the just opened $!_1$-thread. After that new $!_2$-threads (inside the relevant $!_1$-threads) are opened for revisiting (either for access or marking). In order to restrict plays on $\Game$ to the patterns corresponding to Q-machine runs we introduce two classes of variables with differing scopes (cf. Examples 3 and 17).

The first one will comprise "inner" variables that are local to each of the $!_1$-threads. As might be expected they will be SYMBOL, $\mathit{ACCESSED}$, MARKED corresponding to the internal state of the machine's Q-store. Thus, once an ADD block is created it will have fresh and private copies of these variables.

Another group of "outer" variables will range over all $!_1$-threads rather than single $!_1$-threads. They are:

- STATE - corresponding to the state of the simulated Q-machine,
- FIRST - used to distinguish the first ADD operation which initializes the store (initially it is set to 1 and after first ADD will always be equal to 0)
- A and F - used to guide the intermediate stages in the simulation of ADD

and FETCH respectively (initially both are 0 but during the simulation will be incremented to inform about the current stage of the simulation).

Although the state is hidden in plays, we can track the interim values of variables at any time during play by referring to the hidden interactions with copies of ref. The term $M_{\hat{Q}}$ is presented in Figure 5. It has the shape

$$\mathsf{let}\ \vec{outer}\ \mathsf{in}\ (\lambda f.\mathsf{let}\ \vec{inner}\ \mathsf{in}\ \lambda g.(\cdots)).$$

where $(\cdots)$ is a combination of the three terms from Example 37 annotated with code which manipulates and checks values of the variables.

PROOF. (of Lemma 36)

We have to show that complete positions in $[\![M_{\hat{Q}}]\!]$ have the desired shape.

The only initial move is q_0 to which P will reply with $\star_0$ (so $q_0\star_0 \in \mathsf{comp}(M_{\hat{Q}})$). This will be preceded by a hidden interaction interpreting lines 2-5 where the outer variables $STATE, FIRST, A, F$ are initialized respectively to $q_0, 1, 0, 0$.

Let us assume that $s' = q_0 \star_0 s \in [\![M_{\hat{Q}}]\!]$ represents several computational steps of $\hat{Q}$. i.e. s consists of ADD- and FETCH-blocks. We shall prove by induction on the number of simulated steps that, as long as the value of $!STATE$ at the end of s' is *not* in F, all potential extensions of s' to a complete position have to use segments corresponding to further steps of $\hat{Q}$. The case of $!STATE \in F$ will be considered separately at the very end.

It is helpful to identify the following invariants for s':

- the hidden value $!STATE$ at the end of s' corresponds to $\hat{Q}$'s state after the represented steps have taken place;
- the copies of $SYMBOL$,$ACCESSED$, $MARKED$ in ADD blocks have the same values as the corresponding entries of $\hat{Q}$'s Q-store;
- $!A, !F$ are equal to 0 at the end of s';
- $FIRST$ indicates whether the first dummy ADD step (modelling the initialization of the Q-store) was already modelled.

Note that all these are satisfied after the first two moves.

Following s', by the visibility condition, the next O-move can only be one of the following:

(1) q justified by the unique occurrence of $\star_0$,
(2) r justified by an occurrence of $\star$ from an ADD block,

```
let
    STATE = ref(q_0)
    FIRST = ref(1)
    A = ref(0)
    F = ref(0)
 in
    (λf.  let
                □ = [!STATE ∈ Q^A ∧ !A = 0]
                □ = (A := 1)
                SYMBOL = ref(‡)
                ACCESSED = ref(if !FIRST then + else −)
                MARKED = ref(−)
              in
                (λg. if (!A = 1 ∧ !SYMBOL = ‡) then

                          (A := 0;
                            if !FIRST then (FIRST := 0; SYMBOL := †)
                            else (SYMBOL, STATE) := δ^ADD(!STATE);
                          g();
                          [!STATE ∈ F];
                          [!ACCESSED = +])

                     else if (!STATE ∈ Q^F ∧ !F = 0) then

                          ([!ACCESSED = −];
                          F := 1;
                          ACCESSED := +;
                          f();
                          [!F = 2];
                          F := 0;
                          STATE := δ^FETCH(!STATE, !SYMBOL))

                     else if (!F = 1) then

                          ([!ACCESSED = +, !MARKED = −];
                          F := 2;
                          MARKED := +)

                     else Ω_unit)
            end)
end
```

Fig. 5. The term representing a Q-machine.

(3) d_2 justified by a final r_2 from the most recent ADD block.

Recall from Example 37.1 that r_2 corresponds to the $g()$ code, therefore playing d_2 (3) will trigger moves corresponding to lines 20-21. Since we are currently considering only cases when $!STATE \notin F$, P will not be able to reply so (3) cannot extend the position in question.

We analyze the first two cases in turn.

(1) After q is played, moves corresponding to lines 8-12 are played out in the hidden component and we can expect $\star$ to emerge from the interaction (cf. Example 37) if and only if $!STATE \in Q^{\mathrm{ADD}}$ and $!A = 0$ (line 8). Thus, when $!STATE \notin Q^{\mathrm{A}}$, P will have no reply (as required, because then no ADD step is possible). If $!STATE \in Q^{\mathrm{A}}$ then $\star$ will be played and $!A$ will become 1 (line 9). Note also that lines 10-12 will produce fresh copies of $SYMBOL, MARKED$ (initialized to $\ddagger, -$) [5] and of $ACCESSED$ (its initial value depends on $!FIRST$ in the correct way). If $s' = q_0 \star_0$ then, because $q_0 \in Q^{\mathrm{A}}$, $\star$ will be played which will yield the requisite complete position $s' q \star$.

After $\star$ is played, when O subsequently plays q, P will have no reply because $!A$ is now 1 (line 8 blocks the previous response $\star$). Hence, to extend the current position O might play either d_2 or r justified by some occurrence of $\star$. The former would finish the play (since we still have $!STATE \notin F$), so we can focus on the latter possibility. Regardless of the actual justifier, r triggers moves corresponding to the code under λg (lines 14-39), so (because we have $!F = 0$ and $!STATE \in Q^{\mathrm{A}}$) P will reply only if $!A = 1 \wedge !SYMBOL = \ddagger$ on line 14 is satisfied (the reply is then determined by lines 16-21). But for $!SYMBOL = \ddagger$ to hold, the pointer of r must be to an ADD block where $!SYMBOL = \ddagger$. This can only be the case if r points at the preceding move, because in other blocks $!SYMBOL$ has already changed according to the invariants. Therefore, the only O-move leading to an extension is O's r justified by the preceding $\star$ (which is consistent with our design of an ADD block). Then lines 16-18 are played out: $!A$ is reset to 0, $SYMBOL, STATE$ and $FIRST$ are updated as required and r_2 is played (as in Example 37.1) thus completing the creation of a new ADD block.

(2) Because $!F = 0, !A = 0$ after each step, an r move made after a complete step has been simulated is replied to only if $!STATE \in Q^{\mathrm{F}}$ (line 23), which results in interpreting lines 25-31. Supposing r's justifier $\star$ comes from the ith ADD block, P will reply only if $!ACCESSED = -$ in that block (line 25). r_1 is played then (line 28; because of $f()$ like in Example 37.2) and F changes to 1. After r_1 O can see all ADD blocks preceding the ith, some rd pairs left by FETCH and $\overset{\frown}{q r_1}$, where q comes from the ith

[5] We assume that $\ddagger$ is a special symbol, i.e. $\ddagger \notin \Sigma$.

block and r_1 is the last move. This raises many possibilities of moves for O, which we again examine case by case.

(a) d_1 is blocked by line 29.

(b) Any q is blocked by $[!STATE \in Q^{\mathrm{A}}]$ (line 8).

(c) r justified by $\star$ from an ADD block with index j where $j < i$ may trigger d (as in Example 37.3), since we have $!F = 1$ satisfying the test on line 33. However, this will happen if and only if $!ACCESSED = +$ and $!MARKED = -$ in the jth block (line 35) as required by FETCH. The $MARKED$ field of the jth block will be reset to $+$ and F will change to 2.

Next O faces the same possibilities as above. (b) is still blocked by line 8. Line 33 now blocks (c) because we have $!F = 2$. For this reason, d_1 will now trigger d (lines 29-31), but before that $!STATE$ will be updated using $!SYMBOL$ from the ith block and F will be reset to 0, completing a FETCH step.

In the above we have shown that $\mathsf{comp}([\![M_{\hat{Q}}]\!])$ contains $q_0 \star_0$, $q_0 \star_0 q \star$ and possibly other positions but these have to be extensions of plays representing final runs. We show next that the other positions occur only if the final run involved is complete, i.e. when all entries in the Q-store have been accessed.

Consider a position representing a series of runs such that $!STATE \in F$ at its end. O might then play either q (P will not reply because of line 8), or r (P will not reply because line 39 then applies), or d_2. Recall that d_2 is associated with $g()$ (Example 37.1) and it will now trigger d because the test $!STATE \in F$ (line 20) is positive. However, this will happen only if $!ACCESSED = +$ in the relevant block (line 21). Afterwards, the same reasoning applies when subsequent d_2-moves by O refer to other ADD blocks. By playing d_2 repeatedly O and P will in fact verify whether all Q-store entries have been accessed during the simulated final run. Consequently, by Lemma 31, complete positions can only arise when the final run is also complete run. $\square$

Now we can easily reduce the (undecidable) halting problem for Q-machines to a program equivalence query. Observe first that $\mathsf{comp}([\![M]\!]) = \{q_0 \star_0, q_0 \star_0 q \star\}$ for $M \equiv \lambda f.\mathsf{let}\, X = \mathsf{ref}(0)\,\mathsf{in}\ \ (([!X = 0]; X := 1); (\lambda g.\Omega_{\mathsf{unit}}))$.

Theorem 38 *A Q-machine $\hat{Q}$ halts if and only if $M_{\hat{Q}} \cong M$. Second-order* $\mathsf{RML_f}$ *program equivalence is thus undecidable.*

7 Conclusion

We have analyzed an ML-like language $\mathsf{RML_f}$ through its game semantics. $\mathsf{RML_f}$ permits the construction of higher-order functions with local variables $(\mathsf{let}\, v = \mathsf{ref}(0)\, \mathsf{in}\, M : \theta_1 \to \theta_2)$ that persist over multiple applications of the function. In certain cases, made precise in the definition of $\mathsf{RML_f^-}$, that behaviour can be characterized using regular expressions. In all those cases θ is a product of first-order types $\beta_1' \to \beta_2'$ or $\mathsf{int\,ref}$, where the last case can be used to pass a variable out of scope. Consequently, we were able to revisit many program equivalences previously studied in the literature using other techniques, notably, various forms of logical relations [9]. Further increases in the nesting of arrows, e.g. taking $\theta = (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$, lead to loss of regularity or, even worse, make program equivalence undecidable $(\theta = (\mathsf{unit} \to \mathsf{unit}) \to (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit})$. It seems that more decidable cases, like $(\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$, could still be captured with visibly pushdown automata [26], which we have recently used in work on call-by-name games [5].

Acknowledgements

The author gratefully acknowledges support from EPSRC (GR/R88861/01) and St John's College, Oxford.

References

[1] D. R. Ghica, G. McCusker, Reasoning about Idealized Algol using regular expressions, in: Proceedings of ICALP, Vol. 1853 of Lecture Notes in Computer Science, Springer-Verlag, 2000, pp. 103–115.

[2] D. R. Ghica, G. McCusker, The regular language semantics of second-order Idealized Algol, Theoretical Computer Science 309 (2003) 469–502.

[3] S. Abramsky, D. R. Ghica, A. S. Murawski, C.-H. L. Ong, Applying game semantics to compositional software modelling and verification, in: Proceedings of TACAS, Vol. 2988 of Lecture Notes in Computer Science, Springer-Verlag, 2004, pp. 421–435.

[4] C.-H. L. Ong, Observational equivalence of 3rd-order Idealized Algol is decidable., in: Proceedings of IEEE Symposium on Logic in Computer Science, Computer Society Press, 2002, pp. 245–256.

[5] A. S. Murawski, I. Walukiewicz, Third-order Idealized Algol with iteration is decidable, in: Proceedings of FOSSACS, Lecture Notes in Computer Science, Springer-Verlag, 2005, to appear.

[6] D. R. Ghica, A. S. Murawski, C.-H. L. Ong, Syntactic control of concurrency, in: Proceedings of ICALP, Vol. 3142 of Lecture Notes in Computer Science, Springer-Verlag, 2004, pp. 683–694.

[7] D. R. Ghica, Regular-language semantics for a call-by-value programming language, in: Proceedings of MFPS, Vol. 45 of Electronic Notes in Computer Science, Elsevier, 2001.

[8] J. C. Reynolds, Syntactic control of interference, in: Proceedings of POPL, 1978, pp. 39–46.

[9] A. M. Pitts, I. D. B. Stark, Operational reasoning for functions with local state, in: A. D. Gordon, A. M. Pitts (Eds.), Higher-Order Operational Techniques in Semantics, Cambridge University Press, 1998, pp. 227–273.

[10] I. D. B. Stark, Names and higher-order functions, Ph.D. thesis, University of Cambridge Computing Laboratory, Technical Report No. 363 (1995).

[11] S. Abramsky, G. McCusker, Call-by-value games, in: Proceedings of CSL, Vol. 1414 of Lecture Notes in Computer Science, Springer-Verlag, 1997, pp. 1–17.

[12] J. Laird, A game semantics of local names and good variables, in: Proceedings of FOSSACS, Vol. 2987 of Lecture Notes in Computer Science, Springer-Verlag, 2004, pp. 289–303.

[13] S. Abramsky, D. R. Ghica, A. S. Murawski, C.-H. L. Ong, I. D. B. Stark, Nominal games and full abstraction for the nu-calculus, in: Proceedings of LICS, IEEE Computer Society Press, 2004, pp. 150–159.

[14] S. Abramsky, G. McCusker, Game semantics, in: H. Schwichtenberg, U. Berger (Eds.), Logic and Computation, Springer-Verlag, 1998, proceedings of the NATO Advanced Study Institute, Marktoberdorf.

[15] K. Honda, N. Yoshida, Game-theoretic analysis of call-by-value computation (extended abstract), in: Proceedings of ICALP, Vol. 1256 of Lecture Notes in Computer Science, Springer-Verlag, 1997, pp. 225–236.

[16] S. Abramsky, G. McCusker, Games and full abstraction for the lazy lambda calculus, in: Proceedings of IEEE Symposium on Logic in Computer Science, Computer Society Press, 1995, pp. 234–243.

[17] A. S. Murawski, On program equivalence in languages with ground-type references, in: Proceedings of IEEE Symposium on Logic in Computer Science, Computer Society Press, 2003, pp. 108–117.

[18] A. S. Murawski, About the undecidability of program equivalence in finitary languages with state, ACM Transactions on Computational Logic, special LICS'03 issue, to appear.

[19] S. Abramsky, G. McCusker, Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions, in: P. W. O'Hearn, R. D. Tennent (Eds.), Algol-like languages, Birkhaüser, 1997, pp. 297–329.

[20] E. Moggi, Computational lambda-calculus and monads, in: Proceedings of IEEE Symposium on Logic in Computer Science, Computer Society Press, 1989, pp. 14–23.

[21] G. McCusker, Games for recursive types, BCS Distinguished Dissertation, Cambridge University Press, 1998.

[22] S. Abramsky, Algorithmic games semantics: a tutorial introduction, in: H. Schwichtenberg, R. Steinbruggen (Eds.), Proof and System Reliability, Kluwer Academic Publishers, 2001, pp. 21–47, proceedings of the NATO Advanced Study Institute, Marktoberdorf.

[23] J. E. Hopcroft, J. D. Ullman, Introduction to Automata Theory, Languages and Computation, Addison-Wesley, 1979.

[24] E. Post, Formal reductions of the combinatorial decision problem, American Journal of Mathematics 65 (1943) 196–215.

[25] M. L. Minsky, Computation: Finite and Infinite Machines, Prentice-Hall, 1967.

[26] R. Alur, P. Madhusudan, Visibly pushdown languages, in: Proceedings of STOC, 2004, pp. 202–211.