On Program Equivalence in Languages with Ground-Type References

Andrzej S. Murawski*
Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
Andrzej.Murawski@comlab.ox.ac.uk

Abstract

Using game semantics we prove that program equivalence is undecidable in finitary Idealized Algol with active expressions as well as in its call-by-value counterpart.

It is also shown that strategies corresponding to Idealized Algol terms of respectively second, third and higher orders define exactly regular, context-free and recursively enumerable languages.

1. Introduction

Game semantics has emerged as a new semantic paradigm in the quest for a solution to the full abstraction problem for the functional language PCF [2, 11, 16]. Since then it has been employed to construct fully abstract models for a range of programming languages incorporating many other features: first-order references [4, 6], call-by-value evaluation [3, 9], control primitives [12] and general references [1]. Each of these models provides a semantic characterization of program equivalence in the respective programming language, which is given in terms of the intrinsic quotient on strategies. However, for certain languages with ground-type variables, Idealized Algol with active expressions (IA) and a variant of Reduced ML (RML) [4, 3], the quotient can be characterized more explicitly: program equivalence actually coincides with equality of the induced *complete* plays [4]. This makes the languages more amenable to game-semantic analysis and links program equivalence with language equivalence. A number of decidability results for program equivalence have recently been obtained via this route for second- and third-order fragments of finitary (no recursion, finite ground type) Idealized Algol [8, 17] and second-order RML [7]. The papers show that program phrases at the respective orders generate complete plays that are regular or *deterministic* context-free languages. As the equivalence problem for these classes of languages is known to be decidable [10, 19], so is program equivalence in the above-mentioned fragments.

In this paper we show that program equivalence in finitary Idealized Algol is undecidable in general and that this result already applies at fourth order. The proof proceeds by a reduction from the halting problem for a special class of universal queue-equipped machines. This reduction cannot be based on a direct translation, since finitary IA is merely a call-by-name lambda calculus over a finite datatype augmented with imperative constructs (without recursion or loops). Thus, for example, evaluation of any program terminates after a finite number of steps. However, a careful analysis of the induced game semantics reveals that some of the generated positions can be taken to represent computations of various classes of machines. To formalize this observation, we introduce a simple notion of representability (of languages by terms), which we first study at second and third order. It turns out that respectively all regular and context-free languages are representable in these fragments, which demonstrates the accuracy of the characterizations in [8, 17]. At fourth order a Turing-complete class of machines with queues can be modelled in a similar way, which makes program equivalence at that order undecidable. Accordingly, the representable languages are exactly the recursively enumerable ones. The undecidability result for IA can be adapted to the call-by-value language RML by relating the associated game models. Despite some fundamental differences between the frameworks involved in modelling call-by-name and call-by-value, we can identify a fragment of RML whose game semantics inhabits the category of games used to model IA. What is more, for each of the IA terms considered in our representation theorems, we show how to construct an RML term inducing the same semantics. As a result, we can prove RML program equivalence undecidable at fifth order.

Related work. The surprising fact that simple finitary languages can have undecidable observational equivalence has emerged in Ralph Loader's work on call-by-name PCF [13]. His result implied undecidability of equivalence between compact terms and consequently the impossibility

*The author gratefully acknowledges support from UK EPSRC, and Wolfson and St John's Colleges at Oxford.

of presenting fully abstract models for PCF in an effective way. IA in which expressions with side effects are used is not a conservative extension of PCF and Loader's argument cannot be repeated. Actually, it does have an effectively presentable fully abstract model [4], but this does not contradict our result because of the different nature of compactness: even very simple pure lambda terms are not compact in IA. On the technical level our game-based approach is more direct than Loader's, which consisted of a series of syntactic reductions [13]. At the moment it seems tailored to languages with state, but we hope to develop it into a more general argument. Although program equivalence has been studied extensively in a variety of formalisms, to the best of our knowledge, Loader's has been the only undecidability result proved in a finitary setting where the undecidability stems from the use of higher-order procedures.

2. Idealized Algol

Idealized Algol [18] synthesizes functional and imperative programming. Its basis is the simply-typed call-by-name lambda calculus with types $\mathcal{T}$ generated by

$$\mathcal{T} ::= \mathcal{B} \mid var \mid \mathcal{T} \to \mathcal{T} \qquad \mathcal{B} ::= exp \mid com$$

where $\mathcal{B}$ denotes *base* types, com is the type of commands, $exp = \{ x_1, \cdots, x_n \}$ ($n \geq 2$) is the *finite* ground type and var is the type of assignable variables (throughout the paper we use capital letters to refer to them). The order of a type is defined by $order(\mathcal{B}) = order(var) = 0$ and $order(T_1 \to T_2) = max(order(T_1) + 1, order(T_2))$. We will often annotate base subtypes showing their contribution to the overall order, e.g. $(com_2 \to com_1) \to com_0$. By terms of jth order ($j \geq 1$) we mean terms that possess a typing derivation in which the order of types occurring on the left-hand (right-hand) sides of sequents cannot exceed $j - 1$ (respectively j). Figure 1 shows the syntax of finitary IA (additionally, we also admit the standard application and λ-abstraction rules). Equivalently, instead of the $case$ construction, one could consider exp as an initial segment of natural numbers with successor, predecessor and zero test. Given $N : exp$ we shall write $new\, X := N\, in\, M$ for $new\, X\, in\, (X := N; M)$ and assume that X in $new\, X\, in\, M$ is initialized to x_1. We consider the active variant of IA where expressions can be sequenced with commands to yield new expressions possibly with side effects. The syntax thus allows expressions like $(Z := x_5); x_1 : exp$, making it easy to construct terms whose repeated evaluation leads to different values (the more restrictive passive version has been studied e.g. in [3] and can be shown to be a conservative extension of PCF).

Definition 2.1. The term $M : \mathcal{T}$ is said to *approximate* another term $N : \mathcal{T}$ *observationally* ($M \precsim N$) if for all contexts C such that $C[M], C[N]$ are closed terms of type com whenever $C[M]$ converges, so does $C[N]$ (details of the operational semantics can be found in [4]). Two terms $M, N : \mathcal{T}$ are *observationally equivalent* ($M \cong N$) if and only if $M \precsim N$ and $N \precsim M$.

A model is called (equationally) *fully abstract* if it validates precisely the correct equivalences between programs. Observe that if $\cong$ is not decidable, neither is $\precsim$.

Note that neither recursion nor looping is available in the language we consider, but since all compact elements of the fully abstract model [4] are definable in our syntax the addition of recursion would not affect observational equivalence between recursion-free terms. Hence, the results of this paper also apply to deciding equivalences between simple terms in IA with recursion. From this point of view, we show that undecidability already applies to the small subset of nonrecursive programs without the divergent term Ω or the variable constructor $mkvar$. As Ω plays a helpful technical role in our argument, we shall use it throughout the paper. At the end we explain why the same results hold without it.

3. Game models

Game semantics models computation as an exchange of moves between two players called O (Opponent) and P (Proponent). Opponent's moves (O-moves) can be viewed as an environment in which a program is executed, while Proponent's moves (P-moves) represent the behaviour of the interpreted program. There are two kinds of moves: questions (requests for values) and answers (the values themselves). O and P make their moves alternately, adhering to a set of combinatorial rules. A systematic introduction to game semantics can be found in [5]. In what follows we assume that the reader is familiar with such notions as justification, view, visibility and bracketing. The specific models analyzed in this paper are: [4] for IA and [3] for RML. Both provide the following explicit characterization of observational equivalence for the respective languages:

Theorem 3.1. The terms $M_1, M_2 : \mathcal{T}$ (typable in the context Γ) are observationally equivalent if and only if

$$comp([\![\Gamma \vdash M_1 : \mathcal{T}]\!]) = comp([\![\Gamma \vdash M_2 : \mathcal{T}]\!])$$

where $comp(\sigma)$ is the subset of σ consisting of all *non-empty* complete plays (i.e. those in which all questions have been answered) [1].

Before we start working with the models, let us fix some notation. The two moves run and $done$ from $[\![com]\!]$ will

[1] The proof of this theorem for IA can be found in [4]. Because of the definability result (Proposition 12) for finite strategies in [3], the same proof applies to RML.

$$\frac{}{\Gamma \vdash \mathit{skip}, \Omega : \mathit{com}} \qquad \frac{}{\Gamma \vdash x_i : \mathit{exp}} \qquad \frac{\Gamma \vdash M : \mathit{exp} \quad \Gamma \vdash N_i : \mathcal{B} \quad i = 1, \cdots, n}{\Gamma \vdash \mathit{case}(M)[\cdots, N_i, \cdots] : \mathcal{B}} \qquad \frac{\Gamma, X : \mathit{var} \vdash M : \mathcal{B}}{\Gamma \vdash \mathit{new}\, X \,\mathit{in}\, M : \mathcal{B}}$$

$$\frac{\Gamma \vdash M : \mathit{com} \quad \Gamma \vdash N : \mathcal{B}}{\Gamma \vdash M; N : \mathcal{B}} \qquad \frac{\Gamma \vdash M : \mathit{var}}{\Gamma \vdash !M : \mathit{exp}} \qquad \frac{\Gamma \vdash M : \mathit{var} \quad \Gamma \vdash N : \mathit{exp}}{\Gamma \vdash M := N : \mathit{com}} \qquad \frac{\Gamma \vdash M : \mathit{exp} \quad \Gamma \vdash N : \mathit{exp} \to \mathit{com}}{\Gamma \vdash \mathit{mkvar} M N : \mathit{var}}$$

Figure 1. Syntax of finitary (active) Idealized Algol

be abbreviated to r and d. The moves in $[\![exp]\!]$ will be written as q and x_i respectively. If the corresponding subtype has been annotated with a subscript corresponding to its order, for ease of reference we will use an identical superscript to annotate moves of the associated game. Given $x \in X^*$ and $Y \subseteq X$, $x \upharpoonright Y$ is obtained from x by erasing all characters not in Y. $Z \upharpoonright Y$, where $Z \subseteq X^*$, will denote $\{ z \upharpoonright Y \mid z \in Z \}$. Note that $Z \upharpoonright Y$ can be regarded as a homomorphic image of Z with respect to the homomorphism that copies elements of Y and maps each character not in Y to the empty word.

We briefly recall how state is modelled in game semantics [4, 5] and describe how we can take advantage of it. By definition, each maximal position of the game $[\![var]\!]$ is either of the shape $write(x_i)\ ok$ or $read\ x_j$, which corresponds to reference update and dereferencing respectively. In $[\![\Gamma, X : var \vdash M : \mathcal{B}]\!]$ only P can switch to the var component. After he makes a $write(x_i)$ move, O can only play ok, but after $read$, O is free to play any of the moves x_i. Therefore, values read from a free variable are unrelated to values that have been written to it. The binding operation (interpretation of new) does away with that arbitrariness and enforces the standard behaviour of a memory cell: $[\![\Gamma \vdash new\, X\, in\, M : \mathcal{B}]\!]$ consists of precisely those plays of $[\![\Gamma, X : var \vdash M : \mathcal{B}]\!]$ in which each $read$ is followed by the value most recently used in a $write$ move. Additionally, all the moves from $[\![var]\!]$ are erased, making state implicit. However, it will be useful to keep in mind that the (now hidden) state changes as the play unfolds. This can be formalized easily by considering the uncovering of a play with respect to hidden var subgames. Then, if hidden $write(x_i)$ moves associated with some variable X are made between two visible moves m and m', we can say that the value of X at m' has changed accordingly. Note that it is P who makes the $write(x_i)$ moves immediately before binding, so the implicit state cannot change after O-moves. At third order an innocent-like characterization of strategies with explicit state is possible, as it turns out that variable scope coincides with the notion of view [17]. In the sections to follow we concentrate on the game model of IA. We will describe the consequences of our results for the call-by-value setting in the final section of the paper.

4. Representing languages

The hidden state will be used to model the state of various classes of machines.

Definition 4.1. Let $\Sigma = \{ x_1, \cdots, x_n \}$ (i.e. '$\Sigma = exp$') and $L \subseteq \Sigma^*$. We will say that $\Gamma \vdash M_L : \mathcal{T}$ represents L if and only if $L = comp([\![\Gamma \vdash M_L : \mathcal{T}]\!]) \upharpoonright \Sigma$.

$\upharpoonright$ is meant to ignore pointers and select moves originating from the x_i-moves corresponding to occurrences of exp or var in $\mathcal{T}$. We assume that the selected moves are stripped of various tags which may have been used to interpret the disjoint sum in game constructions. Since $comp(\sigma)$ does not contain the empty word ϵ, we can model both $\epsilon \notin L$ and $\epsilon \in L$. Our results can be summarized by

Theorem 4.2. Let $\mathcal{L}_i$ be the set of all languages over Σ that are representable by finitary IA terms of order i.

- $\mathcal{L}_2$ consists of all regular languages.
- $\mathcal{L}_3$ consists of all context-free languages.
- $\mathcal{L}_i$ consists of all recursively enumerable languages for $i \geq 4$.

For each of the three cases our argument will follow a similar pattern. First we will find an appropriate type (which defines sufficiently complex positions) and fix a convention on how transition steps are modelled. This will be the way we want both players to interact in order to simulate the machine in question. By choosing the right term we can constrain P-moves to precisely those that are needed from P provided O follows the convention. As we cannot stop O from making moves not adhering to our representation schemes, we are going to make sure that P is unable to reply in such cases and thus positions in which O fails to cooperate will be prevented from developing into complete ones. This effect will be achieved by inserting the Ω term in various places inside the term so that whenever O deviates from our convention the corresponding strategy will not have a response. The following shorthand expressing 'assertions' will often be used to that end:

$$[condition] \equiv \textbf{if } condition \textbf{ then } skip \textbf{ else } \Omega.$$

The *condition*, e.g. $(!X_1, \cdots, !X_k) \in F$ for some $F \subseteq \Sigma^k$, will always be expressible using the *case* construct (for clarity we will feel free to use *true* and *false* on the understanding that they are represented as two different values). Here is a simple example of how the implicit state can be used to manage O-moves.

Example 4.3. Consider the identity term $\lambda x.x : com_1 \rightarrow com_0$. The corresponding strategy contains the complete position $r^0 r^1 d^1 d^0$. Now consider a similar term $\lambda x.pre; x; post$. For this term, before P can play r^1 the strategy modelling sequential composition will 'play out' *pre* first. If $pre \equiv skip$, then nothing will change and P will immediately play r^1 as before. If $pre \equiv \Omega$ the associated strategy will not have a reply at all. *pre* might also be a side effect, say, $Z := x_3$, in which case P will have to play $write(x_3)$ and only after O follows with ok, will P play r^1. However, if this term is placed within a context that binds Z, the sequence $write(x_3)\, ok$ will be hidden and P will immediately play r^1 after r^0 as for the identity term, although the hidden state will change to x_3 at r^1. The *post* command can change the original identity strategy analogously. For instance, for $post \equiv [!Z = x_3]; (Z := x_5)$, d^0 will be played after d^1 only if x_3 is currently stored in Z (the value of Z at d^0 will then change to x_5); otherwise by playing d^1 O will finish the play.

5. Regular languages

At second order a simplified presentation of game semantics is possible due to the fact that in a legal second-order sequence of moves justification pointers can be reconstructed uniquely. Consequently, positions can be regarded simply as strings of moves. [8] shows that $comp(\sigma)$ is regular for any strategy σ defined by a second-order finitary IA term. Therefore, only regular languages can be represented by such terms, since a homomorphic image of a regular language is also regular [10]. We now show the converse: any regular language $L \subseteq \Sigma^*$ is represented by a closed term

$$M_L : exp \rightarrow (com_2 \rightarrow com_1) \rightarrow com_0$$

in the sense of Definition 4.1. Actually, each of the IA terms used to characterize a class of languages in this paper will have a type of the shape $exp \rightarrow \cdots$. This is quite a natural choice: because of call-by-name evaluation terms of type *exp* may yield different values at different times, which can then be taken to correspond to ordered characters of the represented string.

Let $\langle\, Q, \Sigma, q_0, F, \delta\,\rangle$ be a finite deterministic automaton accepting L, where Q is the finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and $\delta : Q \times \Sigma \rightarrow Q$ is the transition function. M_L will simulate the automaton using a free variable of type $com \rightarrow com$ as an 'iterator'. This has the desirable effect of letting O choose the number of steps so that all possibilities are covered. After the chosen number of transitions have been made, we only need to check whether a final state has been reached. If not, the position is prevented from developing into a complete one. Below we use $STATE$ to represent the state of the automaton, although Q may be larger than the ground type *exp*. In that case we can use several variables to make up for the difference. Thus M_L can be defined as

$$\begin{array}{l}\lambda cn.new\ STATE := q_0\ in \\ \quad (n\,(STATE := \delta(!STATE, c)); [!STATE \in F])\end{array}$$

where $\delta(!STATE, c)$ is a nested (branching on $STATE$ and input symbol respectively) *case* instruction defined in accordance with δ (it is possible to do so in such a way that c will be 'needed' only once: the value of c provided by O can be written to a local variable; this memoization is necessary since repeated queries about c would be visible in the position and consequently a different string would be represented). $comp(\llbracket M_L \rrbracket)$ then comprises all strings of the form

$$r^0 r^1 (r^2 q a_1 d^2) \cdots (r^2 q a_k d^2) d^1 d^0$$

where $a_1 \cdots a_k \in L$ and we have $L = comp(\llbracket M_L \rrbracket) \upharpoonright \Sigma$. Note that the state at d^2 following a_i will correspond exactly to $\delta^*(q, a_1 \cdots a_i)$.

6. Context-free languages

At third order pointers must be included in any representation of positions, e.g. as natural numbers represented by consecutive copies of a special character. Finitary IA terms no longer define regular sets of complete plays then. Instead, they have been shown to be deterministic context-free [17]. Homomorphic images of deterministic context-free languages need not be deterministic context-free, but they are always context-free [10]. So, in the third-order case, one should aim to capture all context-free languages. For that purpose we use a special kind of pushdown automata in which nondeterminism at each state and input symbol has been restricted to the choice between a single push or pop. Additionally, ϵ-transitions are required to be disjoint from transitions using input symbols. Thus, after the choice whether to pop or to push is made, the state (for push) or the state along with the top symbol on the stack (for pop) determine if an input symbol is needed in the next step.

Definition 6.1. A *weak pushdown automaton* is a tuple $\langle\, Q, \Sigma, q_0, F, \Gamma, \delta_\Sigma^{\mathrm{PUSH}}, \delta_\epsilon^{\mathrm{PUSH}}, \delta_\Sigma^{\mathrm{POP}}, \delta_\epsilon^{\mathrm{POP}}\,\rangle$ equipped with a stack, where $Q, \Sigma, q_0, F, \Gamma$ have the standard meaning and $\delta_\Sigma^{\mathrm{PUSH}}, \delta_\epsilon^{\mathrm{PUSH}}, \delta_\Sigma^{\mathrm{POP}}, \delta_\epsilon^{\mathrm{POP}}$ are transition functions

$$
\begin{array}{rll}
pre \equiv & \textbf{if } !STATE \in Q_\Sigma \textbf{ then} & new\ INPUT := c\ in \\
& & ([(!STATE, !INPUT) \in \text{dom } \delta_\Sigma^{\mathrm{PUSH}}]; \\
& & (STATE, STACK) := \delta_\Sigma^{\mathrm{PUSH}}(!STATE, !INPUT)) \\
& \textbf{else} & ([!STATE \in \text{dom } \delta_\epsilon^{\mathrm{PUSH}}]; \\
& & (STATE, STACK) := \delta_\epsilon^{\mathrm{PUSH}}(!STATE))
\end{array}
$$

$$
\begin{array}{rll}
post \equiv & \textbf{if } (!STATE, !STACK) \in (Q \times \Gamma)_\Sigma \textbf{ then} & new\ INPUT := c\ in \\
& & ([(!STATE, !STACK, !INPUT) \in \text{dom } \delta_\Sigma^{\mathrm{POP}}]; \\
& & STATE := \delta_\Sigma^{\mathrm{POP}}(!STATE, !STACK, !INPUT)) \\
& \textbf{else} & ([(!STATE, !STACK) \in \text{dom } \delta_\epsilon^{\mathrm{POP}}]; \\
& & STATE := \delta_\epsilon^{\mathrm{POP}}(!STATE, !STACK))
\end{array}
$$

Figure 2. pre **and** $post$ **code for the simulation of weak pushdown automata.**

representing pushing and popping (with or without accepting an input symbol):

$$
\begin{array}{ll}
\delta_\Sigma^{\mathrm{PUSH}} : Q_\Sigma \times \Sigma \rightharpoonup Q \times \Gamma & \delta_\epsilon^{\mathrm{PUSH}} : Q_\epsilon \rightharpoonup Q \times \Gamma \\
\delta_\Sigma^{\mathrm{POP}} : (Q \times \Gamma)_\Sigma \times \Sigma \rightharpoonup Q & \delta_\epsilon^{\mathrm{POP}} : (Q \times \Gamma)_\epsilon \rightharpoonup Q
\end{array}
$$

where $Q = Q_\Sigma + Q_\epsilon$ and $Q \times \Gamma = (Q \times \Gamma)_\Sigma + (Q \times \Gamma)_\epsilon$ ($X + Y$ denotes the union of two disjoint sets X, Y). We consider acceptance by final state and with empty stack, and assume that initially the stack is empty.[2]

By a routine argument in which various kinds of nondeterminism are replaced with that of weak pushdown automata we obtain:

Theorem 6.2. The languages accepted by weak pushdown automata are precisely the context-free languages.

The rest of this section will be devoted to finding a game representation for weak pushdown machines. We will use the third-order type

$$((com_3 \to com_2) \to com_1) \to com_0$$

to interpret pushing with the sequence $r^2 r^3$ and popping with $d^3 d^2$. Then each sequence of pop and push actions emptying a stack can be modelled by a position of the following shape:

$$\underbrace{r^0 r^1}_{start}\underbrace{r^2 r^3}_{push}\underbrace{r^2 r^3}_{push}\underbrace{r^2 r^3}_{push}\underbrace{d^3 d^2}_{pop}\underbrace{d^3 d^2}_{pop}\underbrace{r^2 r^3}_{push}\underbrace{d^3 d^2}_{pop}\underbrace{d^3 d^2}_{pop}\underbrace{d^1 d^0}_{stop}$$

The moves $r^0 r^1$ initialize the automaton, $d^1 d^0$ indicate that the automaton halted. By the bracketing condition d^3 must always answer the last pending r^3, which corresponds conveniently to the 'last-in first-out' principle. Similarly, d^1 can be played only if all r^2 questions have been answered, i.e. when our representation indicates that the stack is empty. It should be clear that all legal sequences of pop and push instructions leading from an empty stack to an empty stack can be encoded in this way. It so happens that such positions are precisely the complete plays defined by $\lambda f.f(\lambda x.x)$. However, to encode the transitions of a given automaton we should be able to control the order in which pop and push actions can occur in a play.

Example 6.3. The term below defines complete positions corresponding to a series of push actions followed by the same number of pops. Pushing is only allowed as long as the value of $PUSH$ is $true$, the first d^2 (pop) will change it to $false$ blocking further pushes.

$$
\begin{array}{ll}
\lambda f. & new\ PUSH := true\ in \\
& \quad f(\lambda x.[!PUSH = true]; x; PUSH := false)
\end{array}
$$

A similar term of type $exp \to ((com \to com) \to com) \to com$ can be used to represent weak pushdown automata (the code for pre and $post$ is shown in Figure 2):

$$
\begin{array}{l}
\lambda cf.\ new\ STATE := q_0\ in \\
\quad (f(\lambda x.new\ STACK := Z\ in\ pre; x; post); \\
\quad\ [!STATE \in F])
\end{array}
$$

Its game semantics differs from $[\![\lambda f.f(\lambda x.x)]\!]$ in the following way:

- Before P replies with r^3 to r^2 the pre part is 'played out'. Its role is to check whether the current state needs an input symbol for which P can then ask. After O supplies it when needed, it might turn out that the automaton cannot make a push transition from the current state, in which case P will not respond. When a push step can be carried out, $STATE$ will be updated and the pushed symbol will be stored in a new copy of $STACK$. Note that while all $r^2 r^3$ segments share

[2] This is a rather strong convention but it allows us to take the empty word into account when necessary and makes the automata more similar to the class of machines considered in the next section.

a single copy of $STATE$ each of them has a *private* copy of $STACK$, whose value is the symbol pushed on the stack in the corresponding step (thus the players generate an implicit stack of copies of $STACK$, which is used to model the stack of the automaton).

- Before P can play d_2 after O's d_3, *post* checks if a pop step is possible, an input symbol is asked for when necessary and finally $STATE$ will be suitably updated.

Consequently, $!STATE$ at each r^3 and d^2 mirrors the state of the modelled automaton. At the end, P will be able to play d_0 only if $!STATE$ represents a final state, which corresponds to acceptance by final state with empty stack, as required. Hence, each context-free language is representable by a third-order IA term.

7. Recursively enumerable languages

It is well known that two stacks suffice to simulate a Turing machine, but although we showed how to accommodate one, the bracketing condition prevents us from extending the representation scheme to two. Instead we will exploit the fact that the additional move available in fourth-order games can be used to relax the strict rules of access to elements on the stack. In fact, it will turn out possible to access any of them, regardless of their position on the stack, but this arbitrariness has its disadvantages too, because some order will have to be reinstated if machine computations are to be modelled. We will focus on making the liberalized stack into a queue, as state machines with queues are Turing complete (they are essentially the same as tag-systems considered in [14]). First we will introduce an auxiliary class of machines with Q-stores (a nondeterministic generalization of queues) whose computations are representable by complete plays of a fourth-order definable strategy. In addition to stored elements Q-stores contain some bookkeeping information which can be used to verify if a Q-store behaved like a queue. Since this check has a natural game-semantic reading, we will be able to model proper queues as well.

Definition 7.1. A ***Q-store*** stores characters from Σ. Its contents are described by a natural number n (size) and a function $f : \{0, \cdots, n\} \to \Sigma \times \{+, -\} \times \{+, -\}$ (state of a Q-store). The three fields of $f(i)$ will be referred to as $f(i).SYMBOL$, $f(i).ACCESSED$ and $f(i).MARKED$. The first one holds the stored character, the latter two indicate whether the ith entry has already been accessed or marked by the FETCH instruction. Initially, the store is empty: $n = 0$ and $f(0) = (x_1, +, -)$, where x_1 is an irrelevant letter (not considered to be stored but deemed *accessed* and *unmarked*). The state of a Q-store can be updated with two kinds of instructions:

- ADD c adds c to the store. The new state
$$f' : \{0, \cdots, n+1\} \to \Sigma \times \{+, -\}^2$$
is defined by $f \subseteq f'$, $f'(n+1) = (c, -, -)$.

- FETCH is the only access method. It works in the following way: two indices (i, j) such that $0 \leq j < i \leq n$ are chosen if possible; providing that $f(i).ACCESSED = -, f(j).ACCESSED = +, f(j).MARKED = -$ the instruction succeeds and $f(i).ACCESSED, f(j).MARKED$ are both set to $+$. That is to say, FETCH accesses an element not accessed thus far and marks another unmarked but already accessed element in front of the element being accessed. If any of the conditions above is not satisfied (or no suitable pair (i, j) exists) the operation fails. Otherwise, $f(i).SYMBOL$ is returned.

Example 7.2. The state of the Q-store shown below results from the following sequence of operations: ADD a, ADD b, ADD c, FETCH, ADD b, FETCH, ADD a (the FETCH operations chose $(1, 0)$ and $(3, 1)$ respectively). The accessed symbols were a and c.

	0	1	2	3	4	5
$SYMBOL$	x_1	a	b	c	b	a
$ACCESSED$	$+$	$+$	$-$	$+$	$-$	$-$
$MARKED$	$+$	$+$	$-$	$-$	$-$	$-$

Example 7.3. Q-stores operate like queues (first-in first-out) if each FETCH operation chooses i and j to be the index of the first unaccessed record and $i - 1$ respectively. Q-stores resulting from such uses generate characteristic patterns:

0	1	2	3	4	$\cdots$	n
x	a	b	c	d	e	f
$+$	$+$	$+$	$+$	$-$	$-$	$-$
$+$	$+$	$+$	$-$	$-$	$-$	$-$

In particular, the $ACCESSED$ field consists of two contiguous regions containing pluses and minuses respectively. Observe that a deviation from the queue discipline will result in an occurrence of $-$ in between pluses, which cannot be replaced later. For example, FETCH will never access b in the Q-store from Example 7.2, because no suitable element will be available for marking. Therefore, given the state of a Q-store one can immediately tell whether it acted like a queue. We incorporate Q-stores into a simple machine model next.

Definition 7.4. A ***Q-machine*** is a tuple $\langle Q, \Sigma, q_0, F, \delta^{\mathrm{ADD}}, \delta^{\mathrm{FETCH}} \rangle$ equipped with a Q-store:

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

A Q-machine makes nondeterministic transitions from the initial state and stops when a final state is reached. An unsuccessful execution of a FETCH instruction is treated as 'divergence'. Q-machines in which the Q-store is guaranteed to operate like a queue are deterministic and will be called ***proper***.

Theorem 7.5. Proper Q-machines can simulate Turing machines and thus have an undecidable halting problem. □

The fact that queue-based machines with state can be universal was known a long time ago as such machines are closely related to Post systems [14]. For technical convenience we will assume that the initial state of a Q-store results from a special ADD (executed once at the very beginning) which introduces the unmarked accessed symbol x_1 but does not affect the state.

Let us now return to the encoding of pushdown automata considered previously. If we work with the fourth-order type $(((com_4 \to com_3) \to com_2) \to com_1) \to com_0$ stack traces like $r^0r^1r^2r^3r^2r^3d^3d^2r^2r^3d^3d^2d^3d^2d^1d^0$ will be in 1-1 correspondence with plays from $comp(\llbracket \lambda h.h(\lambda f.f\,\Omega)\rrbracket)$. Note that Ω prevents P from replying after O uses r^4, so there is no complete play in which r^4 is used. Of course we are going to replace Ω with a different term. The additional move r^4 turns out to make a big difference as it can be interpreted as an access to arbitrary elements already stored in a stack (rather than just the one on top) enabling us to model FETCH. ADD operations will be represented in the same way as push moves for pushdown automata: O plays r_2 and P replies with r_3. Let us consider the sequence representing several ADD instructions:

$$r^0r^1\underbrace{r^2r^3}_{0}\underbrace{r^2r^3}_{1}\cdots\underbrace{r^2r^3}\cdots\underbrace{r^2r^3}\cdots\underbrace{r^2r^3}_{n}$$

The representation of FETCH will be divided into three stages: choosing the symbol to access (i), choosing the symbol to mark (j), and finally updating the state of the machine and its Q-store. In the last stage all moves from the previous two stages will be hidden from O-view in order to prevent O from using them as justifiers later.

• An attempt to access a symbol will correspond to playing r^4r^3

$$r^0r^1\underbrace{r^2r^3}_{0}\underbrace{r^2r^3}_{1}\cdots\underbrace{r^2r^3}\cdots\underbrace{r^2r^3}_{i}\cdots\underbrace{r^2r^3}_{n}r^4r^3$$

This behaviour is consistent with the strategy interpreting $\lambda h.h(\lambda f.f(f\,\Omega))$. Note that due to Ω, the game will be over, if O subsequently plays r^4 justified by the r^3 that P has just played.

• After r^3 is played as above, O is only able to see blocks r^2r^3 indexed by $j \le i$ in his view; marking will correspond to playing r^4d^4, where r^4 is justified by r^3 from the jth block:

$$r^0r^1\underbrace{r^2r^3}_{0}\underbrace{r^2r^3}_{1}\cdots\underbrace{r^2r^3}_{j}\cdots\underbrace{r^2r^3}_{i}\cdots\underbrace{r^2r^3}_{n}r^4r^3r^4d^4$$

This time the last move coincides with the responses defined by $\lambda h.h(\lambda f.f\;skip)$.

• Finally we would like the players to answer the questions raised in the first point with d^3d^4:

$$r^0r^1\underbrace{r^2r^3}_{0}\underbrace{r^2r^3}_{1}\cdots\underbrace{r^2r^3}_{j}\cdots\underbrace{r^2r^3}_{i}\cdots\underbrace{r^2r^3}_{n}r^4r^3r^4d^4d^3d^4$$

which will hide all actions from the previous two steps from O-view. When O plays next he will only be able to play moves justified by either r^1 or the r^3 moves introduced by ADD actions (there will also be r^4d^4 segments in the view but they do not open any new possibilities for O).

As before, we are going to use the variable $STATE$ to track the state of the machine. At r^3 (for ADD) and at the second d^4 (for FETCH) $!STATE$ will always be equal the state of the machine after it executes all the represented steps. Once a final state is reached O and P will play as many d^3d^2 blocks as necessary to complete the play with d^1d^0 (P should not be able to respond to O's attempts to carry out an ADD or FETCH step then). The term M representing a given Q-machine in the way just described has the following shape:

$$\begin{array}{l}
\lambda h.new\ STATE := q_0, ACCESS := false,\\
\qquad MARKING := false, FIRST := true\\
\quad in\\
\qquad (h(\lambda f.new\ SYMBOL, ACCESSED := -,\\
\qquad\qquad\qquad MARKED := -\\
\qquad\qquad in\\
\qquad\qquad\quad ([!STATE \in Q^{\mathrm{ADD}}]; add;\\
\qquad\qquad\quad f\,([!STATE \in Q^{\mathrm{FETCH}}]; fetch);\\
\qquad\qquad\quad [!STATE \in F]));\\
\qquad [!STATE \in F])
\end{array}$$

The term combines the behaviour exhibited by $\lambda h.h(\lambda f.f\;skip)$ and $\lambda h.h(\lambda f.f(f\,\Omega))$ and 'switches' between them (cf.$fetch$) according to the values of variables tracking various stages of the simulation. The add and $fetch$ subroutines are shown in Figure 3. Note that after each r^3 new 'local' copies of $SYMBOL$,$ACCESSED$ and $MARKED$ are produced, while $STATE$,$ACCESS$,$MARKING$, $FIRST$ are global (initialized at r^1).

Theorem 7.6. $s \in comp(\llbracket M\rrbracket)$ if and only if s represents a successful computation of the associated Q-machine.

Proof. By case analysis we show that $\llbracket M\rrbracket$ contains only positions corresponding to histories of the Q-machine. Due

$$
\begin{array}{lrl}
add \equiv & \textbf{if } !FIRST = true \textbf{ then} & (SYMBOL := x_1; ACCESSED := +; FIRST := false) \\
& \textbf{else} & (STATE, SYMBOL) := \delta^{\mathrm{ADD}}(!STATE) \\[1em]
fetch \equiv & \textbf{if } !ACCESS = false \textbf{ then} & ([!ACCESSED = -]; ACCESS := true; (f\ \Omega); \\
& & [!MARKING = true]; \\
& & ACCESS := false; MARKING := false; ACCESSED := +; \\
& & STATE := \delta^{\mathrm{FETCH}}(!STATE, !SYMBOL)) \\
& \textbf{else} & ([!ACCESSED = +, !MARKED = -, !MARKING = false]; \\
& & MARKING := true; MARKED := +)
\end{array}
$$

Figure 3. add **and** $fetch$

to the specially positioned occurrences of assertions inside M, P will never be able to reply after O deviates from the representation scheme or tries to model an illegal step, e.g. a FETCH step when $!STATE \notin Q^{\mathrm{FETCH}}$. Furthermore, because of the occurrences of $[!STATE \in F]$, complete positions can only be reached after a final state is entered. □

Ultimately we are interested only in the traces in which the store is actually used like a queue so that proper Q-machines can be captured. Recall that the complete plays from Theorem 7.6 end with a series of segments d^3d^2 followed by d^1d^0. We have already observed that the queue regime generates a characteristic contiguous pattern of $+$ and $-$ in a Q-store and its violation leaves a $-$ between pluses in the $ACCESSED$ fields. Fortunately, this condition can be verified during play in the final sequence of answer-moves.

Theorem 7.7. For each *proper* Q-machine there exists a closed term M which defines complete plays corresponding to successful runs of the machine, i.e. the machine does not halt if and only if $comp([\![M]\!]) = \varnothing$ (or equivalently $M \cong \lambda h.\Omega$).

Proof. The order in which the answers d^3d^2 are played at the end of positions simulating Q-machines corresponds to scanning the Q-store from right to left. We introduce a new global variable $FLAG$ (initially equal to $false$) in the term from Theorem 7.6 so that during the 'scan' it can be changed to $true$ when the rightmost record with $!ACCESSED = +$ is encountered. Afterwards, if a record satisfying $!ACCESSED = -$ exists, the position should not be completed. To achieve that it suffices to replace

$$f\,([!STATE \in Q^{\mathrm{FETCH}}]; fetch); [!STATE \in F]$$

with

$$
\begin{array}{l}
f\,([!STATE \in Q^{\mathrm{FETCH}}]; fetch); [!STATE \in F]; \\
[!FLAG \Rightarrow (!ACCESSED = +)]; \\
\textbf{if } !ACCESSED = + \textbf{ then } FLAG := true
\end{array}
$$

□

Since proper Q-machines are Turing-strong, we have:

Theorem 7.8. Observational equivalence in finitary IA is undecidable (at fourth order). □

To represent any recursively enumerable language, we need to make some provision for inputing symbols so that the queue can be initialized with any string before the machine tries to accept it. We will modify the term from Theorem 7.7 to achieve this. For a start, we add $\lambda c.$ in front of the term to get a term of type

$$exp \to (((com \to com) \to com) \to com) \to com.$$

Next we will introduce some occurrences of c inside it. The new term will still represent the corresponding Q-machine, but the simulation will now begin with an input stage during which input symbols are added to the queue. Each of the input steps will be modelled by r^2r^3 in the same way as ADD actions, but $!STATE$ will remain q_0. We must let O decide when to finish the input stage, because there can be no bound on the length of input. O will be able to do so by playing r^4. This will set a global variable (call it $INPUT$), whose initial value is $true$, to $false$. Once this happens, the simulation proper will commence in the old way. To implement all the changes, we replace $(STATE, SYMBOL) := \delta^{\mathrm{ADD}}(!STATE)$ in add with

$$
\begin{array}{l}
\textbf{if } !INPUT = true \textbf{ then } SYMBOL := c \\
\textbf{else } (STATE, SYMBOL) := \delta^{\mathrm{ADD}}(!STATE)
\end{array}
$$

and, in order to allow O to finish the input phase by r^4 (justified by any of the r^3 moves introduced before), we replace $[!STATE \in F]; fetch$ with

$$
\begin{array}{l}
\textbf{if } !INPUT = true \textbf{ then } INPUT := false \\
\qquad\qquad\qquad\quad \textbf{else } ([!STATE \in F]; fetch)
\end{array}
$$

The resultant term defines complete plays whose projections onto Σ correspond exactly to the strings that the proper Q-machine accepts. Hence, any recursively enumerable language is representable in finitary IA. As $comp(\llbracket M \rrbracket)$ is recursively enumerable for any IA term M, we have proved the last part of Theorem 4.2.

8. Call-by-value

In this section we turn to finitary RML, which is essentially the call-by-value lambda calculus augmented with imperative constructs and ground-type ML-style references (there is no equality test for references though). A fully abstract game semantics for RML has been presented in [3]. Call-by-value types are interpreted as families of games, terms as families of strategies. Sum games define the required strong monad $(T, \eta, ^*)$ [15]: given a family $\{ A_i \}_{i \in I}$, $T(\{ A_i \}_{i\in I}) = \{ \sum_{i\in I} !A_i \}$. Let $\llbracket \mathcal{T} \rrbracket_v$ be the call-by-value denotation of $\mathcal{T}$ (the subscript n will be used for call-by-name). RML typing judgments $x_1 : A_1, \cdots, x_k : A_k \vdash B$ define morphisms between $\Pi_{i=1}^{k} \llbracket A_i \rrbracket_v$ and $\{ T(\llbracket B \rrbracket_v) \}$. Function types are modelled by $\llbracket A \to B \rrbracket_v = \llbracket A \rrbracket_v \to T(\llbracket B \rrbracket_v)$. For instance, $\llbracket exp \rrbracket_v = \{ 1_{x_i} \}_{1 \le i \le n}$, $\llbracket com \rrbracket_v = \{ 1 \}$ and $\llbracket com \to B \rrbracket_v = T(\llbracket B \rrbracket_v)$ (1 is the empty game). Although the call-by-value game model seems much more complicated, some RML types can be interpreted using singleton families of games, i.e. essentially games, just like for IA. For example, we have $\llbracket var \rrbracket_v = \llbracket var \rrbracket_n$, $T(\llbracket \mathcal{B} \rrbracket_v) = \llbracket \mathcal{B} \rrbracket_n$ for $\mathcal{B} = exp, com$ and $\llbracket T_k \rrbracket_n = \llbracket T_{k+1} \rrbracket_v$, where $T_0 = com$ and $T_{k+1} = T_k \to com$ for $k \ge 0$. We list more similarities between the two frameworks in Figure 4 in the form of rules which relate call-by-name ($\llbracket \Gamma \vdash M : \mathcal{T} \rrbracket_n$) and call-by-value ($\llbracket \Gamma' \vdash M' : \mathcal{T}' \rrbracket_v$) denotations. Note that some of them apply to terms with *base* types only. In what follows we show that each of the strategies discussed previously in the context of IA is also denoted by an RML term. The undecidability of RML observational equivalence follows then easily as for IA.

Observe that imperative constructs are modelled in the same way in both frameworks. Therefore, in order to find an RML term defining the same strategy as one of the IA terms we have considered, it suffices to work out the call-by-value counterpart of the underlying lambda term and then decorate it with commands in an analogous way. For example, to obtain a call-by-value variant of Example 6.3 we observe that $\llbracket x : com \vdash x \rrbracket_n = \llbracket x : com \to com \vdash x\, skip \rrbracket_v$ and after applying the obvious rules from Figure 4 deduce

$$\begin{aligned}&\llbracket f : T_3 \vdash new\ P := true\ in \\ &\qquad f(\lambda x.[!P = true]; (x\, skip); P := false) : com \rrbracket_v = \\ &\llbracket f : T_2 \vdash new\ P := true\ in \\ &\qquad\qquad f(\lambda x.[!P = true]; x; P := false) : com \rrbracket_n\end{aligned}$$

Let us now recall that the IA term $M : T_4$ used to interpret a given proper Q-machine had the following shape:

$$\lambda h.new \cdots in\ h(\lambda f.new \cdots in\ c_1; f(c_2; \textbf{if} \cdots \\ \textbf{then}\ c_3; f\Omega; c_4\ \textbf{else}\ c_5); c_6); c_7.$$

The equality below can be derived when one starts off with $\llbracket \Omega : com \rrbracket_n; \eta_{\llbracket com \rrbracket} = \llbracket \lambda x.\Omega : com \to com \rrbracket_v$ and uses $\llbracket \Gamma \vdash N : \mathcal{T} \rrbracket_v = \llbracket \Gamma, x : com \vdash N : \mathcal{T} \rrbracket_v$:

$$\llbracket h : T_3 \vdash h(\lambda f.f(f\,\Omega)) : com \rrbracket_n = \\ \llbracket h : T_4 \vdash h(\lambda f.f(\lambda x.f(\lambda y.\Omega))) : com \rrbracket_v.$$

Because $\llbracket \Gamma_{var} \vdash c_i : com \rrbracket_n = \llbracket \Gamma_{var} \vdash c_i : com \rrbracket_v$ for $i = 1, \cdots, 7$, where Γ_{var} consists of var-identifiers, we will have $\llbracket M : T_4 \rrbracket_n; \eta_{\llbracket T_4 \rrbracket_n} = \llbracket M' : T_5 \rrbracket_v$ for the RML term M' shown below:

$$\lambda h.new \cdots in\, h(\lambda f.new \cdots in\, c_1; f(\lambda x.c_2; \textbf{if} \cdots \\ \textbf{then}\ c_3; f(\lambda y.\Omega); c_4\ \textbf{else}\ c_5); c_6); c_7.$$

Theorem 8.1. For each proper Q-machine there exists an RML term M' such that the machine does not halt if and only if $M' \cong \lambda h.\Omega$. Observational equivalence in finitary RML is undecidable. □

For call-by-value this result holds at order five. Analogously, the types of all RML counterparts of the IA terms considered in this paper will have types of order by one higher. Hence, regular, context-free and recursively enumerable languages can be represented at order three, four, five respectively (input corresponds to an argument of type $com \to exp$ rather than exp). Since it is not known what language classes are captured by RML terms of order three and four, the results are only representation rather than characterization results. Similarly, the program equivalence problem for RML at these orders remains open.

$\Omega : com$ has played an important role in our encodings. Without it we have to find a way to handle complete positions that do not correspond to any machine runs (normally they would not arise thanks to occurrences of Ω). Fortunately, it is possible to modify the terms used previously so that the undesirable complete positions are distinguished from the good ones, e.g. by returning a result which identifies the position: $true$ if it represents a machine run and $false$ otherwise (the terms will then have type of the form $\cdots \to exp$ instead of $\cdots \to com$). To this end we can introduce a global variable OK (initialized to $true$), replace each occurrence of $\Omega : com$ by $OK := false$ and arrange for P to play $!OK$ as the final move. Then the successful runs correspond to plays ending in $true$. What is more, the question whether any such run exists can be stated as an equivalence query (with a term in which $false$ is always returned instead of $!OK$). Thus the undecidability results also apply to the Ω-free subsets of finitary IA and RML.

References

[1] S. Abramsky, K. Honda, and G. McCusker. Fully abstract game semantics for general references. In *Proceedings of IEEE Symposium on Logic in Computer Science*. Computer Society Press, 1998.

$$\frac{[\![A]\!]_n = [\![A']\!]_v}{[\![x : A \vdash x : A]\!]_n ; \eta_{[\![A]\!]_n} = [\![x : A' \vdash x : A']\!]_v} \qquad \frac{\begin{array}{c}[\![\Gamma \vdash M_1 : com]\!]_n = [\![\Gamma' \vdash M_1' : com]\!]_v \\ [\![\Gamma \vdash M_2 : \mathcal{B}]\!]_n = [\![\Gamma' \vdash M_2' : \mathcal{B}]\!]_v\end{array}}{[\![\Gamma \vdash M_1 ; M_2 : \mathcal{B}]\!]_n = [\![\Gamma' \vdash M_1' ; M_2' : \mathcal{B}]\!]_v}$$

$$\frac{[\![\Gamma \vdash M : exp]\!]_n = [\![\Gamma' \vdash M' : exp]\!]_v \quad [\![\Gamma \vdash N_i : \mathcal{B}]\!]_n = [\![\Gamma' \vdash N_i' : \mathcal{B}]\!]_v}{[\![\Gamma \vdash case(M)[\cdots, N_i, \cdots] : \mathcal{B}]\!]_n = [\![\Gamma' \vdash case(M')[\cdots, N_i', \cdots] : \mathcal{B}]\!]_v}$$

$$\frac{[\![\Gamma \ \vdash \ M : exp]\!]_n = [\![\Gamma' \ \vdash \ M' : exp]\!]_v}{[\![\Gamma, X : var \ \vdash \ (X := M) : com]\!]_n = [\![\Gamma', X : var \ \vdash \ (X := M') : com]\!]_v}$$

$$\frac{[\![\Gamma, X : var \ \vdash \ M : \mathcal{B}]\!]_n = [\![\Gamma', X : var \ \vdash \ M' : \mathcal{B}]\!]_v}{[\![\Gamma \ \vdash \ new \ X \ in \ M : \mathcal{B}]\!]_n = [\![\Gamma' \ \vdash \ new \ X \ in \ M' : \mathcal{B}]\!]_v}$$

$$\frac{[\![\Gamma]\!]_n = [\![\Gamma']\!]_v}{[\![\Gamma, X : var \vdash !X : exp]\!]_n = [\![\Gamma', X : var \vdash !X : exp]\!]_v} \quad \frac{[\![\Gamma, x : A \vdash M : \mathcal{B}]\!]_n = [\![\Gamma, x : A' \vdash M' : \mathcal{B}]\!]_v}{[\![\Gamma \vdash \lambda x.M : A \to \mathcal{B}]\!]_n ; \eta_{[\![A \to \mathcal{B}]\!]_n} = [\![\Gamma \vdash \lambda x'.M' : A' \to \mathcal{B}]\!]_v}$$

$$\frac{[\![\Gamma \vdash M : A \to \mathcal{B}]\!]_n ; \eta_{[\![A \to \mathcal{B}]\!]_n} = [\![\Gamma' \vdash M' : A' \to \mathcal{B}]\!]_v \qquad [\![\Gamma \vdash N : A]\!]_n ; \eta_{[\![A]\!]_n} = [\![\Gamma' \vdash N' : A']\!]_v}{[\![\Gamma \vdash MN : \mathcal{B}]\!]_n = [\![\Gamma' \vdash M'N' : \mathcal{B}]\!]_v}$$

The last rule signifies the fact that call-by-value application coincides with call-by-name when the two terms involved are values: $f = \langle\, \eta_A, \eta_B \,\rangle ; \tau^l_{A,B} ; f^*$ holds, where $f : A \times B \to TC$ and $\tau_l^{A,B}$ is the canonical "left pairing" morphism [15].

Figure 4. Relating call-by-name and call-by-value semantics

[2] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Information and Computation*, 163:409–470, 2000.

[3] S. Abramsky and G. McCusker. Call-by-value games. In *Proceedings of CSL*, volume 1414 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 1997.

[4] S. Abramsky and G. McCusker. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. In P. W. O'Hearn and R. D. Tennent, editors, *Algol-like languages*. Birkhaüser, 1997.

[5] S. Abramsky and G. McCusker. Game semantics. In H. Schwichtenberg and U. Berger, editors, *Logic and Computation*. Springer-Verlag, 1998. Proceedings of the 1997 Marktoberdorf Summer School.

[6] S. Abramsky and G. McCusker. Full abstraction for Idealized Algol with passive expressions. *Theoretical Computer Science*, 227:3–42, 1999.

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

[8] D. R. Ghica and G. McCusker. Reasoning about idealized algol using regular expressions. In *Proceedings of ICALP*, volume 1853 of *Lecture Notes in Computer Science*, pages 103–115. Springer-Verlag, 2000.

[9] K. Honda and N. Yoshida. Game-theoretic analysis of call-by-value computation (extended abstract). In *Proceedings of ICALP*, volume 1256 of *Lecture Notes in Computer Science*, pages 225–236. Springer-Verlag, 1997.

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

[11] J. M. E. Hyland and C.-H. L. Ong. On Full Abstraction for PCF: I. Models, observables and the full abstraction problem, II. Dialogue games and innocent strategies, III. A fully abstract and universal game model. *Information and Computation*, 163(2):285–408, 2000.

[12] J. Laird. *A semantic analysis of control*. PhD thesis, University of Edinburgh, 1998.

[13] R. Loader. Finitary PCF is not decidable. *Theoretical Computer Science*, 266(1-2):341–364, 1997.

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

[15] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[16] H. Nickau. *Hereditarily Sequential Functionals: A Game-Theoretic Approach to Sequentiality*. PhD thesis, Universität-Gesamthochschule-Siegen, 1996.

[17] 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.

[18] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North Holland, 1978.

[19] C. Stirling. Decidability of DPDA equivalence. *Theoretical Computer Science*, 255:1–31, 2001.