About the undecidability of program equivalence in finitary languages with state

ANDRZEJ S. MURAWSKI
Oxford University Computing Laboratory

We show how game semantics can be employed to prove that program equivalence in finitary Idealized Algol with active expressions is undecidable. We also investigate a notion of representability of languages by terms and show that finitary Idealized Algol terms of respectively second, third and higher orders define exactly regular, context-free and recursively enumerable languages.

Categories and Subject Descriptors: F.2.3 [**Theory of Computation**]: Logics and Meanings of Programs—*Semantics of Programming Languages*; F.1.1 [**Theory of Computation**]: Computation by Abstract Devices—*Models of Computation*

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Game semantics, Idealized Algol, program equivalence

1. INTRODUCTION

Equivalence is one the most fundamental notions in computer science. Whenever a computational model is defined, one of the first steps following its definition is the introduction of a suitable notion of equivalence. Capturing equivalence in more abstract terms has become a challenge for semanticists. Game semantics has emerged as a new paradigm from that quest. Originally, it provided a semantic account of equivalence in the functional language PCF [Abramsky et al. 2000; Hyland and Ong 2000; Nickau 1996], but since then has been extended to accommodate more complicated features like ground-type references [Abramsky and McCusker 1997b; 1999], call-by-value evaluation [Abramsky and McCusker 1997a; Honda and Yoshida 1997], control primitives [Laird 1998] and general references [Abramsky et al. 1998]. Each of these models provides a characterization of program equivalence in the respective programming language, given in terms of a quotient on strategies. However, for certain languages, notably Idealized Algol with active expressions (IA), the quotient can be characterized explicitly: program equivalence actually coincides with equality of the induced *complete* plays. This makes the languages more amenable to game-semantic analysis and connects program equivalence with language equivalence. A number of decidability results for program

The author gratefully acknowledges support from UK EPSRC and St John's College, Oxford.
Author's address: OUCL, Wolfson Building, Parks Road, Oxford OX1 3QD, United Kingdom.
Email: Andrzej.Murawski@comlab.ox.ac.uk

equivalence have recently been obtained via this route for second- and third-order fragments of finitary (no recursion, finite ground type) IA [Ghica and McCusker 2000; Ong 2002]. The papers show that program phrases at the respective orders generate sets of complete plays which are regular or *deterministic* context-free languages. As the equivalence problem for these classes of languages is known to be decidable [Hopcroft and Ullman 1979; Sénizergues 2001], so is program equivalence.

In this paper we show that program equivalence in finitary IA is undecidable in general and that this already applies to fourth-order terms. 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 geometric patterns which 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) and study it at various orders. This leads to the following result. We write $\mathcal{L}_i$ for the class of languages representable by terms of ith order.

THEOREM 1.1. —*$\mathcal{L}_2$ is the class of regular languages.*

—*$\mathcal{L}_3$ is the class of context-free languages.*

—*$\mathcal{L}_i$ is the class of recursively-enumerable languages for any $i \geq 4$.*

The surprising fact that simple finitary languages can have undecidable observational equivalence has emerged in Ralph Loader's work on call-by-name PCF [Loader 1997]. His result implied undecidability of equivalence between compact terms and, consequently, the impossibility of constructing an effectively presentable fully abstract model for PCF. Idealized Algol in which expressions with side effects are used is not a conservative extension of PCF, so Loader's argument cannot be repeated. In fact, the fully abstract game model of IA is effectively presentable [Abramsky and McCusker 1997b]. This does not contradict our result though, since even very simple finitary IA terms, e.g. λ-terms, are not compact in IA.

Other undecidability results for subrecursive languages found in the literature usually rely on some infinitary feature, e.g. an infinite storage device [Ibarra et al. 2000] or an infinite ground datatype [Constable and Muchnick 1972]. Our result demonstrates that the behaviour of higher-order terms, even finitary ones, can be equally complicated.

2. IDEALIZED ALGOL

Idealized Algol [Reynolds 1978] synthesizes functional and imperative programming. Its basis is the simply-typed call-by-name lambda calculus with types generated according to the following grammar

$$T ::= \quad \mathbf{com} \quad | \quad \mathbf{exp} \quad | \quad \mathbf{var} \quad | \quad T \to T$$

where **com** is the type of commands, **exp** is the type of expressions and **var** is the type of mutable variables. We consider a finitary variant IA of Idealized Algol and assume that **exp** is *finite* and equal to $\{0, \cdots, N\}$ for some $N > 0$ (i.e. **exp**

$$\Delta \vdash i : \mathbf{exp} \qquad \Delta \vdash \mathit{skip}, \Omega : \mathbf{com} \qquad \frac{\Delta \vdash M : \mathbf{exp} \quad \Delta \vdash M_i : B \quad i = 0, \cdots, N}{\Delta \vdash \mathbf{case}(M)[\cdots, M_i, \cdots] : B}$$

$$\Delta, x : T \vdash x : T \qquad \frac{\Delta, x : T \vdash M : T'}{\Delta \vdash \lambda x.M : T \to T'} \qquad \frac{\Delta \vdash M : T \to T' \quad \Delta \vdash N : T}{\Delta \vdash MN : T'}$$

$$\frac{\Delta \vdash M : \mathbf{com} \quad \Delta \vdash N : B}{\Delta \vdash M; N : B} \qquad \frac{\Delta \vdash M : \mathbf{var}}{\Delta \vdash !M : \mathbf{exp}} \qquad \frac{\Delta \vdash M : \mathbf{var} \quad \Delta \vdash N : \mathbf{exp}}{\Delta \vdash M := N : \mathbf{com}}$$

$$\frac{\Delta, X : \mathbf{var} \vdash M : B}{\Delta \vdash \mathbf{new}\ X\ \mathbf{in}\ M : B} \qquad \frac{\Delta \vdash M : \mathbf{exp} \to \mathbf{com} \quad \Delta \vdash N : \mathbf{exp}}{\Delta \vdash \mathbf{mkvar}\, MN : \mathbf{var}}$$

$$B ::= \mathbf{com} \,|\, \mathbf{exp}$$

Fig. 1. Syntax of finitary *active* Idealized Algol

contains at least two different values). Similarly, variables of type **var** can only store values from that range.

Type inference rules of IA are given in Figure 1. IA has expressions with side-effects as well as the variable constructor **mkvar** which can be used to define "objects" of type **var** with arbitrary read and write "methods". Equivalently, instead of **case** one could use successor, predecessor and zero test. Neither recursion nor iteration is available in the language, but we have the divergent constant $\Omega : \mathbf{com}$ (definable by either). We often write $\mathbf{new}\ X := M\ \mathbf{in}\ N$ to mean $\mathbf{new}\ X\ \mathbf{in}\ ((X := M); N)$. Syntactic identity will be denoted by $\equiv$.

Given $\Theta = \{x_1 : \mathbf{var}, \cdots, x_n : \mathbf{var}\}$, a Θ-state is a function $s : \Theta \to \{0, \cdots, N\}$. For terms-in-context $\Theta \vdash M : T$ we define evaluation rules in Figure 2 as relations $M, s \Downarrow c, s'$ where s, s' are Θ-states and c depends on T. If $T = \mathbf{com}$ then $c \equiv \mathit{skip}$, $c \in \{0, \cdots, N\}$ for $T = \mathbf{exp}$, $c \equiv x_i$ or $c \equiv \mathbf{mkvar}\, MN$ for $T = \mathbf{var}$, and c is a λ-abstraction when T is a function type.

We shall write $M, s \Downarrow$ if $M, s \Downarrow c, s'$ for some c, s'. When M is closed, this will simply become $M \Downarrow$. For example, since there are no rules for Ω, we do not have $\Omega \Downarrow$.

Definition 2.1. The term $\Delta \vdash M : T$ is said to *approximate* another term $\Delta \vdash N : T$ if for all IA-contexts $\mathcal{C}$ such that $\mathcal{C}[M], \mathcal{C}[N]$ are closed terms of type **com** if $\mathcal{C}[M] \Downarrow$ then $\mathcal{C}[N] \Downarrow$. Then we write $M \sqsubseteq\!\!\!\sim N$.

Two terms $\Delta \vdash M, N : T$ are *observationally equivalent* (written as $M \cong N$) if $M \sqsubseteq\!\!\!\sim N$ and $N \sqsubseteq\!\!\!\sim M$.

Observe that if $\cong$ is not decidable, neither is $\sqsubseteq\!\!\!\sim$.

Remark 2.2. The notions of program approximation and equivalence will not change if contexts with recursion are added. This is because all compact elements of the fully abstract model of IA are already definable by finitary terms. Thus our results are immediately relevant to full IA and from that perspective we study approximation and equivalence between simple terms of the full language.

It also turns out that the inclusion of **mkvar** does not affect program equivalence (provided the conditional is extended to type **var**); perhaps surprisingly, this does

$$c, s \Downarrow c, s \qquad \frac{M, s \Downarrow i, s' \quad M_i, s' \Downarrow c, s''}{\mathbf{case}(M)[M_0, \cdots, M_N], s \Downarrow c, s''} \qquad \frac{M, s \Downarrow skip, s' \quad N, s' \Downarrow c, s''}{M; N, s \Downarrow c, s''}$$

$$\frac{M, s \Downarrow \lambda x.M', s' \quad M'[N/x], s' \Downarrow c, s''}{MN, s \Downarrow c, s''} \qquad \frac{M, (s \,|\, x \mapsto 0) \Downarrow c, (s' \,|\, x \mapsto i)}{\mathbf{new}\ x\ \mathbf{in}\ M, s \Downarrow c, s'}$$

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

$$\frac{M, s \Downarrow x, s'}{!M, s \Downarrow s'(x), s'} \qquad \frac{M, s \Downarrow \mathbf{mkvar}\, M'N', s' \quad N', s' \Downarrow i, s''}{!M, s \Downarrow i, s''}$$

Fig. 2. Operational semantics

not apply to program approximation [McCusker 2001].

3. GAME SEMANTICS

Game semantics views computation as an exchange of moves between two players called *Opponent* (O) and *Proponent* (P). The play takes place on an *arena* associated with the type of the interpreted program. Proponent always follows a *strategy* determined by the interpreted program, whereas Opponent is free to make any legal move at any time, thus representing any of the possible environments in which the program may be run. Arenas and strategies are respectively the objects and the morphisms in game models. In what follows we briefly review the definition of the game model of IA [Abramsky and McCusker 1997b; 1998].

3.1 Arenas

Definition 3.1. 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 projections.

—$\vdash_A$ is a binary relation on M_A, called *enabling*, satisfying
 —if $m \vdash_A n$ for no m then $\lambda_A(n) = (O, Q)$,
 —if $m \vdash_A n$ then $\lambda_A^{OP}(m) \neq \lambda_A^{OP}(n)$,
 —if $m \vdash_A n$ then $\lambda_A^{QA}(m) = Q$.

If $m \vdash_A n$ we say that m *enables* n. When m is a question and n is an answer we say that n is a possible answer to m. We shall write I_A for the set of all moves of A which have no enabler; such moves are called *initial*. Note that initial moves must be Opponent questions. In what follows we write o and p respectively for O-moves and P-moves and use q and a for questions and answers.

The *product* $(A \times B)$ and *arrow* $(A \Rightarrow B)$ arenas are defined by:

$$\begin{aligned} M_{A\times B} &= M_A + M_B \\ \lambda_{A\times B} &= [\lambda_A, \lambda_B] \\ \vdash_{A\times B} &= \vdash_A + \vdash_B \end{aligned}$$

$$\begin{aligned} M_{A\Rightarrow B} &= M_A + M_B \\ \lambda_{A\Rightarrow B} &= [\langle \lambda_A^{PO}, \lambda_A^{QA} \rangle, \lambda_B] \\ \vdash_{A\Rightarrow B} &= \vdash_A + \vdash_B + \{(b,a) \mid b \in I_B \text{ and } a \in I_A\} \end{aligned}$$

where $\lambda_A^{PO}(m) = O$ if and only if $\lambda_A^{OP}(m) = P$.

The simplest arena is the empty one, denoted by 1, for which $M_1 = \emptyset$ ($\lambda_I, \vdash_I$ are then uniquely determined). In arenas used to interpret the base types all questions are initial and all answers are P-moves:

Arena	O-questions	possible P-answers
$[\![\mathbf{com}]\!]$	*run*	*done*
$[\![\mathbf{exp}]\!]$	q	n
$[\![\mathbf{var}]\!]$	*read*	n
	write(n)	*ok*

In the table n stands for any $0 \le n \le N$. Note that $[\![\mathbf{var}]\!] = [\![\mathbf{exp}]\!] \times [\![\mathbf{com}]\!]^{(N+1)}$.

3.2 Positions

A *justified sequence* in arena A is a finite sequence of moves of A equipped with pointers. The first move must be initial (and thus an O-move) but after that no consecutive moves can belong to the same player (so, it is O who begins and then P and O take turns). Moreover, the initial O-move has no pointer, but each subsequent move n must have a unique pointer to an earlier occurrence of a move m such that $m \vdash_A n$. Then we say that n is (explicitly) justified by m or, when n is an answer, that n answers m. In order for a justified sequence to become a "legal" play two further conditions must be met.

—The *visibility* condition stipulates that the justifier of any move must appear in the view of the player who considers making the move. O-views (denoted by $\llcorner s \lrcorner$ where s is an even-length position) and P-views ($\ulcorner s \urcorner$ for s of odd length) are defined symmetrically as follows:

$$\begin{aligned} \llcorner \epsilon \lrcorner &= \epsilon & \ulcorner o \urcorner &= o \\ \llcorner\, s o \cdots p\, \lrcorner &= \llcorner s \lrcorner\, o\, p & \ulcorner\, s p \cdots o\, \urcorner &= \ulcorner s \urcorner\, p\, o \end{aligned}$$

where ϵ denotes the empty sequence. Observe that a player can hide some recent history of the play from the view of the other player by making a move that is justified by another move played much earlier.

—The *bracketing* condition states that whenever an answer is played it must answer the last unanswered question.

Definition 3.2. A justification sequence s over A is called a *play* or *position* if it satisfies visibility and bracketing.

The set of plays on arena A is denoted by P_A. For example, we have $P_{A\times B} = P_A + P_B$. Given a set of plays $P \subseteq P_A$, we define $P^{\circledast}$ as the set of all interleavings of finitely many plays from P. Then each play from $P_{A\Rightarrow B}$ is an interleaving of a play from P_B with a sequence from $P_A^{\circledast}$. Moreover, whenever action moves between A and B (or between the interleaved plays in A) inside a play from $P_{A\Rightarrow B}$, the move that makes the switch belongs to P.

3.3 Strategies

A strategy is a specification of Player's responses to some of the plays at which P is allowed to make a move.

Definition 3.3. A *strategy* σ on arena A (henceforth written as $\sigma : A$) is a prefix-closed subset of P_A consisting of positions of even length.

Given an odd-length position s (i.e. one ending in an O-move) we will say that a strategy σ has a response at s if it contains a sequence extending s. For IA only deterministic strategies need to be considered, such strategies have at most one response at any position. For any arena A we have the identity strategy $\mathsf{id}_A : A \Rightarrow A$ which tells P to copy O-moves between the left and the right instances of A. Since the initial move in $A \Rightarrow A$ can only be in the right copy of A, id_A first copies from right to left and then alternates between the "left-to-right" and "right-to-left" modes.

Two strategies $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$ can be composed by considering their possible interactions in the shared arena B. Moves in B are subsequently hidden yielding a sequence of moves in A and C. Recall that a play on $B \Rightarrow C$ might contain several plays on B (intuitively, this corresponds to a function using its argument a number of times). Hence, to make the strategies interact properly, we will have to consider interaction of τ with copies of σ.

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. The set of all such sequences will be denoted by $inter(A, B, C)$. 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). Finally, define $u \upharpoonright A, C$ to be 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 $m_B \in M_B$ extend the pointer to the initial move in C which was pointed to from m_B. Then the composite strategy is defined by

$$\sigma;\tau = \{u \upharpoonright A, C \mid u \in inter(A, B, C),\ u \upharpoonright A, B \in \sigma^{\circledast},\ u \upharpoonright B, C \in \tau\}.$$

Arenas and strategies define a category $\mathcal{G}$ in which morphisms between two arenas A_1 and A_2 are strategies on $A_1 \Rightarrow A_2$. This category is cartesian closed: products are defined by the $\times$ construction and pairing amounts to taking sums of strategies (up to a canonical embedding into the disjoint sum). The empty arena is the terminal object. Because for any arenas A, B, C the arenas $A \times B \Rightarrow C$ and $A \Rightarrow (B \Rightarrow C)$ are identical up to the associativity of the disjoint sum, when we define currying and uncurrying using the canonical associativity maps, the $\Rightarrow$ construction will generate the requisite cartesian closure.

Example 3.4. In the evaluation strategy $\mathsf{ev}_{A,B} : (A \Rightarrow B) \times A \Rightarrow B$, obtained by uncurrying $\mathsf{id}_{A\Rightarrow B}$, O must always start by playing a move in the right copy of B. P makes the same move in the left copy then. Afterwards O may make a move in A which will initiate copying between the two copies of A or he could play a move in B and continue copying B-moves. A switch between the two modes, copying of A-moves or copying of B-moves, may only occur if O is to make a move in the $A \Rightarrow B$ subgame. We will often refer to the move that initiates copying in the A component as a "query about the argument".

In order to define the fully abstract model, the category $\mathcal{G}$ must be quotiented by a simple equivalence relation. Let us call a play *complete* if all questions therein have answers (observe that complete plays are precisely the maximal ones). Intuitively, complete plays are representations of terminating computations. Given a strategy τ we write $\mathsf{comp}(\tau)$ for the set of all *non-empty* complete plays of τ. Then we define

$$\tau_1 \approx \tau_2 \qquad \text{if and only if} \qquad \mathsf{comp}(\tau_1) = \mathsf{comp}(\tau_2).$$

Next we show how to interpret the syntax of IA in $\mathcal{G}$ and thus also $\mathcal{G}/\approx$.

3.4 Game model

A term-in-context $x_1 : T_1, \cdots, x_n : T_n \vdash M : T$ is interpreted by a strategy (denoted by $[\![x_1 : T_1, \cdots, x_n : T_n \vdash M : T]\!]$) for the game

$$[\![T_1]\!] \times \ldots \times [\![T_n]\!] \Rightarrow [\![T]\!].$$

The constants are modelled as follows

$$[\![\Omega : \mathbf{com}]\!] = \{\epsilon\} \qquad [\![\mathit{skip} : \mathbf{com}]\!] = \{\epsilon, \mathit{run}\ \mathit{done}\} \qquad [\![i : \mathbf{exp}]\!] = \{\epsilon, q\ i\}.$$

The rest of the syntax is interpreted compositionally using special strategies which we list below. Because all these strategies have plays that are prefixes of their complete plays, we will only mention the complete plays. We use subscripts to indicate the game from which moves originate.

For instance, $[\![\Delta \vdash C_1; C_2 : B]\!] = \langle\, [\![\Delta \vdash C_1 : \mathbf{com}]\!], [\![\Delta \vdash C_2 : B]\!]\, \rangle; \mathsf{seq}_B$. where $\mathsf{seq}_B : [\![\mathbf{com}]\!]_2 \times [\![B]\!]_1 \Rightarrow [\![B]\!]_0$ is defined by $q_0\ \mathit{run}_2\ \mathit{done}_2\ q_1\ a_1\ a_0$ for any q, a such that $q \vdash_{[\![B]\!]} a$. Conditionals are interpreted using $\mathsf{case}_B : [\![\mathbf{exp}]\!]_{N+1} \times [\![B]\!]_0 \times \ldots \times [\![B]\!]_N \Rightarrow [\![B]\!]$ defined by $q\ q_{N+1}\ i_{N+1}\ q_i\ a_i\ a$ for $i = 0, \cdots, N$. For assignment the strategy $\mathsf{assign} : [\![\mathbf{var}]\!]_2 \times [\![\mathbf{exp}]\!]_1 \Rightarrow [\![\mathbf{com}]\!]_0$ generated by $\mathit{run}_0\ q_1\ n_1\ \mathit{write}(n)_2\ \mathit{ok}_2\ \mathit{done}_0$ is used, while $\mathsf{deref} : [\![\mathbf{var}]\!]_1 \Rightarrow [\![\mathbf{exp}]\!]_0$ containing the complete positions $q_0\ \mathit{read}_1\ i_1\ i_0$, where $0 \le i \le N$, models dereferencing. Finally, we have $[\![\Delta \vdash \mathbf{new}\ X\ \mathbf{in}\ M : B]\!] = \Lambda_X([\![\Delta, X : \mathbf{var} \vdash M : B]\!]); \mathsf{cell}_B$ where Λ_X means currying and $\mathsf{cell}_B : ([\![\mathbf{var}]\!]_2 \Rightarrow [\![B]\!]_1) \Rightarrow [\![B]\!]_0$ has complete plays of the shape

$$q_0\ q_1\ (\mathit{read}_2\ 0_2)^*\ (\sum_{i=0}^{N}(\mathit{write}(i)_2\ \mathit{ok}_2\ (\mathit{read}_2\ i_2)^*))^*\ a_1\ a_0.$$

We take a closer look at this interpretation. Note that in any play of $[\![\Delta, X : \mathbf{var} \vdash M : B]\!]$ the moves from the $[\![\mathbf{var}]\!]$ component are a sequence of segments $\mathit{read}\ i$ and $\mathit{write}(i)\ \mathit{ok}$ but, since O is making the moves following read, there is no logical connection between the read and write moves. In the interaction sequence of

$[\![\Delta, X : \mathbf{var} \vdash M : B]\!]$ and cell_B this changes and only sequences with the "memory cell" behaviour participate in the interaction. Subsequently, the $[\![\mathbf{var}]\!]$ component is hidden and state changes become invisible.

Remark 3.5. For our purposes however it will turn out useful to bring back the explicit nature of state and bear in mind that the hidden state changes as the play unfolds. Thus, if a hidden $write(i)$ move associated with some variable X is made between two visible moves m_1 and m_2 (those that have not been hidden by interaction with cell) we will say that the value of X at m_2 is i or has changed to i. Note that the above can only happen if m_1 is an O-move and m_2 is a P-move. In this way, we can talk about values of variables at visible moves, although the semantics hides them. This ability to trace value of variables will be used to relate our encodings with machine models.

The game interpretation of IA terms outlined above gives rise a fully abstract model [Abramsky and McCusker 1997b].

THEOREM 3.6 FULL ABSTRACTION. *For $\Delta \vdash M_1, M_2 : T$ we have*

$$M_1 \sqsubsetsim M_2 \iff \mathsf{comp}([\![M_1]\!]) \subseteq \mathsf{comp}([\![M_2]\!]).$$

4. STRATEGIES AND LANGUAGES

For the purpose of representing arbitrary languages over an alphabet Σ we identify elements of Σ with values of type $\mathbf{exp}$ and assume that $\Sigma = \{x_0, \cdots, x_N\}$ is actually $\{0, \cdots, N\}$.

Given a IA type T and $s \in P_{[\![T]\!]}$ we write $s \upharpoonright \Sigma$ for the string in Σ^* obtained from s by erasing all moves *not* of the form $\{0, \cdots, N\}$ and ignoring the pointers. The selected moves are contributed by occurrences of $\mathbf{exp}$ and $\mathbf{var}$ in T.

Strictly speaking, $s \upharpoonright \Sigma$ is not really a subsequence of s, because the moves must be extracted from the disjoint sums used to construct the arenas and pointer-related information must be deleted. If justified sequences are encoded as words over some suitable alphabet, e.g. $M_{[\![T]\!]} \cup \{\diamond\}$, where strings of $\diamond$'s are used to represent pointers, then $\cdot \upharpoonright \Sigma$ can be regarded as an erasure homomorphism between $(M_{[\![T]\!]} \cup \{\diamond\})^*$ and Σ^*. Recall that by the visibility condition the justifier of each move is present in the relevant player-view. Typically, the $\diamond$'s are used to represent the distance between the justified move and the justifier in that view.

Given $P \subseteq P_{[\![T]\!]}$ we write $P \upharpoonright \Sigma$ for $\{s \upharpoonright \Sigma \mid s \in P\}$.

Definition 4.1. $\Delta \vdash M : T$ represents $L \subseteq \Sigma^*$ if and only if

$$L = \mathsf{comp}([\![\Delta \vdash M : T]\!]) \upharpoonright \Sigma.$$

Because $\mathsf{comp}(\sigma)$ does not contain the empty play by definition, we can faithfully represent both $\epsilon \in L$ and $\epsilon \notin L$.

Example 4.2. Let

$$M \equiv \lambda x_1 x_2.\mathbf{new}\ X\ \mathbf{in}\ \mathbf{case}(x_1)[X := x_2, \cdots, X := x_2] : \mathbf{exp}_2 \to \mathbf{exp}_1 \to \mathbf{com}_0.$$

We have:

$$\mathsf{comp}([\![M]\!]) = \{run_0\, q_1 \diamond i_1 \diamond q_2 \diamond\diamond\diamond\ j_2 \diamond\ done_0 \diamond\diamond\diamond\diamond\diamond \mid i, j \in \{0, \cdots, N\}\}$$

and thus M represents exactly all two-character words from Σ^*.

We are going to relate languages representable by terms to some well-known classes of languages, as already spelt out in Theorem 1.1. The classification will be based on the order of the terms.

The order of a type is defined by $\mathit{order}(\mathbf{com}) = \mathit{order}(\mathbf{exp}) = \mathit{order}(\mathbf{var}) = 0$ and $\mathit{order}(T_1 \to T_2) = \mathit{max}(\mathit{order}(T_1) + 1, \mathit{order}(T_2))$. A term-in-context $\Delta \vdash M : T$ has order j if all typing judgments $x_1 : T_1', \cdots, x_n : T_n' \vdash M : T'$ used to derive $\Delta \vdash M : T$ satisfy $\mathit{order}(T') \leq j$ and $\mathit{order}(T_h') < j$ for each $h = 1, \cdots, n$.

For instance, terms of order 0 can only be built from constants and thus are constants themselves. Then Ω represents the empty language, *skip* corresponds to $\{\epsilon\}$ whereas i represents $\{i\}$. Thus, at order 0, only the simplest languages can be represented: the empty one and the singleton languages (containing either the empty word or a letter of the alphabet).

Let $\mathcal{L}_i$ be the class of languages representable by terms of order i. In the sections to follow we characterize $\mathcal{L}_i$ for $i \geq 1$. The main difficulty in each case will be to find a way of representing various machine actions in game semantics. Regardless of the class of machines involved our argument will adhere to the following pattern. We will first identify an appropriate type (one that defines sufficiently complex positions) and fix a convention on how transition steps are modelled. This will be the way we expect both players to play in order to represent machine runs. By choosing the right term we can constrain P-moves to precisely those that are needed from P, but not so for O-moves. Games normally allow O to explore more positions than required in the simulation and we will have to make sure that the undesirable ones cannot develop into complete plays. 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

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

The *condition* (e.g. $x \in F$ for some $F \subseteq \Sigma$) 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 two fixed and different elements of **exp**). The following example illustrates the idea of blocking or delaying O-moves.

Example 4.3. Consider the identity term $\lambda x.x : \mathbf{com}_1 \to \mathbf{com}_0$. The corresponding strategy contains the complete position $\mathit{run}_0\, \mathit{run}_1\, \mathit{done}_1\, \mathit{done}_0$. Now consider a similar term $\lambda x.\mathbf{new}\ Y, Z\ \mathbf{in}\ \mathit{pre}; x; \mathit{post}$. For this term, before P can play run_1 the strategy modelling sequential composition will 'play out' *pre* first. If $\mathit{pre} \equiv \mathit{skip}$, then nothing will change and P will immediately play run_1 as before. If $\mathit{pre} \equiv \Omega$ the associated strategy will not have a reply at all. *pre* might also be a side effect, say, $Z := 3$, in which case P will play run_1 but we will keep in mind that the hidden moves $\mathit{write}(3)\mathit{ok}$ have been played, so the (hidden) value of Z at run_1 will change to 3.

The *post* command will affect done_0. For instance, for $\mathit{post} \equiv [!Z = 8]; Y := 5$, done_0 will be played after done_1 only if 8 has previously been stored in Z by the *pre* code (the hidden value of Y at done_0 will then change to 5). If a different value is stored in Z, P will have no reply and the position will not be completed.

5. REGULAR LANGUAGES

Justification pointers are needed to capture the complicated interactions of higher-order terms with their environment, but in plays generated by second-order terms they can always be uniquely reconstructed. Consequently, second-order strategies can be represented simply as sequences of moves. The following result can be shown using this pointer-free, yet faithful, representation [Ghica and McCusker 2000].

THEOREM 5.1. $\mathsf{comp}([\![x_1 : T_1, \cdots, x_n : T_n \vdash M : T]\!])$ *is a regular language over* $M_{[\![T_1]\!]\times\ldots\times[\![T_n]\!]\Rightarrow[\![T]\!]}$ *for any 2nd order finitary* IA *term* $x_1 : T_1, \cdots, x_n : T_n \vdash M : T$.

Consequently, all languages $L \subseteq \Sigma^*$ representable by second-order terms must be regular, since homomorphic images of regular languages are also regular (see, for instance, the classic textbook [Hopcroft and Ullman 1979]). We shall show the converse: any regular language L can be defined by a term $M_L : \mathbf{exp} \rightarrow (\mathbf{com}_2 \rightarrow \mathbf{com}_1) \rightarrow \mathbf{com}_0$. M_L will actually implement a finite state machine accepting L, so its relationship with L will be obvious. The choice of the type for M_L is quite natural. Because of call-by-name evaluation terms of type $\mathbf{exp}$ may evaluate to different values of $\mathbf{exp}$ at different times so their history of use defines strings. The type $\mathbf{com}_2 \rightarrow \mathbf{com}_1$ in turn is a good substitute for iteration: a variable $n : \mathbf{com} \rightarrow \mathbf{com}$ represents any procedure and, accordingly, its argument may be used an arbitrary number of times (in game semantics this will correspond to letting O query the argument any number of times). If the argument carries out a transition step as a side-effect, n uses will simulate n transition steps.

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. Then M_L can be taken to be

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

where $\delta(!STATE, c)$ is built by nesting $\mathbf{case}$ terms which branch on $!STATE$ and c respectively (this arrangement guarantees that c is needed only once). Note that whenever a play from $[\![M_L]\!]$ visits the subgame $[\![\mathbf{com}]\!]_2 \Rightarrow [\![\mathbf{com}]\!]_1$, O can play either r_2 or d_1. The former is a query about the argument of n which begins the simulation of another transition step, the latter completes the transition phase and triggers a response by P only if a final state has been reached. Thus we get

$$\mathsf{comp}([\![M_L]\!]) = \{\quad r_0\, r_1\, (r_2\, q\, a_1\, d_2) \cdots (r_2\, q\, a_k\, d_2)\, d_1\, d_0 \quad | \quad a_1 \cdots a_k \in L\},$$

where r, d stand for *run*, *done*. In particular, $L = \mathsf{comp}([\![M_L]\!]) \restriction \Sigma$, and we get:

THEOREM 5.2. $\mathcal{L}_2$ *is the class of regular languages.*

Before we go on to context-free languages we briefly discuss the first-order case. M_L is not a first-order term, because arguments of first-order terms can be of type $\mathbf{com}, \mathbf{exp}$ or $\mathbf{var}$. In fact, first-order terms generate only finite sets of complete plays (the regular expressions from [Ghica and McCusker 2000] never use Kleene star for such terms). Because these sets come from strategies, they can be viewed as trees branching on O-moves. However, nontrivial branchings are only possible when O is to play a move from $\{0, \cdots, N\}$, since in other cases only one move will be available to O (due to the shape of first-order arenas). Consequently, languages from $\mathcal{L}_1$ can also be represented as trees, whose branches are annotated with characters from Σ and paths from the root to the leaves determine the words of the language. This mode of representation implies the so-called *prefix* property [Hopcroft and Ullman 1979]: if $v, w \in L$ and w is a prefix of v then $w = v$.

Conversely, all finite languages with the prefix property are representable by first-order terms. Let $L \subseteq \Sigma^*$ be such a language. Then $L = \emptyset$, $L = \{\epsilon\}$ or $L = \bigcup_{i=0}^{N}(iL_i)$ where each L_i is finite, satisfies the prefix property, and at least one of L_i is not empty. We define the representing term $x : \mathbf{exp} \vdash M_L : \mathbf{com}$ as follows:

$$\begin{array}{rcl} M_\emptyset & \equiv & \Omega \\ M_{\{\epsilon\}} & \equiv & \mathit{skip} \\ M_{\bigcup_{i=0}^{N}(iL_i)} & \equiv & \mathbf{case}(x)[M_{L_0}, \cdots, M_{L_N}] \end{array}$$

Thus, $\mathcal{L}_1$ consists of finite languages with the prefix property.

6. CONTEXT-FREE LANGUAGES

Positions generated by third-order terms can no longer be identified with the underlying sequences of moves and information about pointers has to be included in any faithful representation, e.g. as described in Example 4.2. The induced sets of complete plays can then be shown to be deterministic context-free [Ong 2002].

THEOREM 6.1. $\mathsf{comp}(\llbracket x_1 : T_1, \cdots, x_n : T_n \vdash M : T\rrbracket)$ *is a deterministic context-free language over* $M_{\llbracket T_1\rrbracket\times\ldots\times\llbracket T_n\rrbracket\Rightarrow\llbracket T\rrbracket} \cup \{\diamond\}$ *for any 3rd order finitary* IA *term* $x_1 : T_1, \cdots, x_n : T_n \vdash M : T$.

As homomorphic images of deterministic context-free languages are always context-free and not necessarily deterministic context-free [Hopcroft and Ullman 1979], at third order we should aim to capture the context-free languages.

To that end we use a special kind of a pushdown automaton in which nondeterminism at each state and input symbol has been restricted to the choice between a single push or pop. Additionally, we require ϵ-productions to be disjoint from other transitions. 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) will determine whether an input symbol is processed in the next step.

Definition 6.2. A *weak pushdown automaton* is a tuple

$$\langle\, Q, \Sigma, q_0, F, \Gamma, \delta_\Sigma^{\mathrm{PUSH}}, \delta_\epsilon^{\mathrm{PUSH}}, \delta_\Sigma^{\mathrm{POP}}, \delta_\epsilon^{\mathrm{POP}} \,\rangle$$

where Q, Σ, q_0, F have the same meaning as for finite automata, Γ is the stack alphabet and $\delta_\Sigma^{\mathrm{PUSH}}, \delta_\epsilon^{\mathrm{PUSH}}, \delta_\Sigma^{\mathrm{POP}}, \delta_\epsilon^{\mathrm{POP}}$ are transition functions representing pushing and popping (with or without accepting an input symbol):

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

where $Q = Q_\Sigma + Q_\epsilon$ and $Q \times \Gamma = (Q \times \Gamma)_\Sigma + (Q \times \Gamma)_\epsilon$. We assume that initially the stack is empty and the automata accept strings by final state *and* empty stack. This is a rather strong convention but it will allow us to take the empty word into account when necessary.

THEOREM 6.3. *The languages accepted by weak pushdown automata are precisely the context-free languages.*

PROOF. The proof relies on replacing various kinds of nondeterminism in standard pushdown automata with that of weak pushdown automata. A pushdown automaton (as defined in [Sipser 1997]) is a tuple $\langle Q, \Sigma, q_0, F, \Gamma, \delta \rangle$, where Q, Σ, q_0, F have the same meaning as in the definition of finite automata, Γ is the stack alphabet and

$$\delta : Q \times \Gamma_\epsilon \times \Sigma_\epsilon \to \wp(Q \times \Gamma_\epsilon)$$

is the transition function ($X_\epsilon = X \cup \{\epsilon\}$, $\wp(X)$ denotes the powerset of X). We consider acceptance by final state with empty stack and assume that the stack is empty when the automaton starts from q_0. The above definition allows for simultaneous popping and pushing ($(q', Z') \in \delta(q, Z, x)$) as well as transitions that do not have any effect on the stack ($(q', \epsilon) \in \delta(q, \epsilon, x)$). These two actions can be simulated by respectively a pop ($(q', \epsilon) \in \delta(q, Z, x)$) followed by a push ($(q', Z') \in \delta(q, \epsilon, x)$) and a push followed by a pop so, without any loss of expressivity, we can focus on simulating automata where the transition function consists of two maps $\delta^{\mathrm{POP}}, \delta^{\mathrm{PUSH}}$ responsible for pushing and popping respectively:

$$\begin{aligned} \delta^{\mathrm{PUSH}} &: Q \times \Sigma_\epsilon \longrightarrow \wp(Q \times \Gamma) \\ \delta^{\mathrm{POP}} &: Q \times \Gamma \times \Sigma_\epsilon \longrightarrow \wp(Q). \end{aligned}$$

Next we show that the multivaluedness of these functions can also be restricted so that we deal with two partial functions

$$\begin{aligned} \delta^{\mathrm{PUSH}} &: Q \times \Sigma_\epsilon \rightharpoonup Q \times \Gamma \\ \delta^{\mathrm{POP}} &: Q \times \Gamma \times \Sigma_\epsilon \rightharpoonup Q. \end{aligned}$$

To that end we will have to introduce a number of fresh states (denoted by $\bullet$ in figures) and modify the transition function as shown below.

$$q \xrightarrow{x/Z'} q' \quad \text{and} \quad q \xrightarrow{x,Z} q'$$

stand respectively for $(q', Z') \in \delta^{\mathrm{PUSH}}(q, x)$ and $q' \in \delta^{\mathrm{POP}}(q, Z, x)$.

—$\delta^{\mathrm{PUSH}}(q, x) = \{(q_1, Z_1), (q_2, Z_2), (q_3, Z_3), \cdots\}$ can then be replaced by

$$\begin{tikzcd}
q \arrow[r, "x/Z"] & \bullet \arrow[r, "\epsilon/Z"] \arrow[d, "\epsilon,Z"] & \bullet \arrow[r, "\epsilon/Z"] \arrow[d, "\epsilon,Z"] & \bullet \arrow[r, "\epsilon/Z"] \arrow[d, "\epsilon,Z"] & \cdots \\
& \bullet \arrow[d, "\epsilon/Z_1"] & \bullet \arrow[d, "\epsilon,Z"] & \bullet \arrow[d, "\epsilon,Z"] & \\
& q_1 & \bullet \arrow[d, "\epsilon/Z_2"] & \bullet \arrow[d, "\epsilon,Z"] & \\
& & q_2 & \bullet \arrow[d, "\epsilon/Z_3"] & \\
& & & q_3 &
\end{tikzcd}$$

—$\delta^{\mathrm{POP}}(q, Z, x) = \{q_1, q_2, q_3, \cdots\}$ can be replaced by

$$\begin{tikzcd}
q \arrow[r, "{x,Z}"] & \bullet \arrow[r, "\epsilon/Z"] & \bullet \arrow[r, "\epsilon/Z"] \arrow[d, "{\epsilon,Z}"] & \bullet \arrow[r, "\epsilon/Z"] \arrow[d, "{\epsilon,Z}"] & \bullet \arrow[r, "\epsilon/Z"] \arrow[d, "{\epsilon,Z}"] & \cdots \\
 & & q_1 & \bullet \arrow[d, "{\epsilon,Z}"] & \bullet \arrow[d, "{\epsilon,Z}"] & \\
 & & & q_2 & \bullet \arrow[d, "{\epsilon,Z}"] & \\
 & & & & q_3 &
\end{tikzcd}$$

Note that an automaton defined by two partial functions may still have a choice between using an input symbol and making an ϵ-move when it makes a push or pop at a fixed state. This sort of nondeterminism can be eliminated too so that whenever $\delta^{\mathrm{PUSH}}(q, x)$ (respectively $\delta^{\mathrm{POP}}(q, Z, x)$) is defined, $\delta^{\mathrm{PUSH}}(q, \epsilon)$ (respectively $\delta^{\mathrm{POP}}(q, Z, \epsilon)$) is not. This can be done as follows.

—Given $q \in Q$ consider all $a \in \Sigma$ such that $\delta^{\mathrm{PUSH}}(q, a)$ is defined and assume $\delta^{\mathrm{PUSH}}(q, a) = (q_a, Z_a)$. Suppose $\delta^{\mathrm{PUSH}}(q, \epsilon) = (q_\epsilon, Z_\epsilon)$. The same effect can be achieved with:

$$\begin{tikzcd}
q \arrow[r, "\epsilon/Z_\epsilon"] & \bullet \arrow[r, "\epsilon/Z_\epsilon"] & \bullet \arrow[r, "\epsilon/Z_\epsilon"] \arrow[dr, "{\epsilon,Z_\epsilon}"'] & \bullet \arrow[r, "{\epsilon,Z}"] & \bullet \arrow[r, "{\epsilon,Z}"] & \bullet \arrow[r, "{\epsilon,Z}"] & \bullet \arrow[r, "a/Z_a"] & q_a \\
 & & & q_\epsilon & & & &
\end{tikzcd}$$

—Given $(q, Z) \in Q \times \Gamma$ consider all $a \in \Sigma$ such $\delta^{\mathrm{POP}}(q, Z, a)$ is defined and assume $\delta^{\mathrm{POP}}(q, Z, a) = q_a$. Suppose $\delta^{\mathrm{POP}}(q, Z, \epsilon) = q_\epsilon$. The same effect is achieved with:

$$\begin{tikzcd}
q \arrow[r, "{\epsilon,Z}"] & \bullet \arrow[r, "\epsilon/Z"] & \bullet \arrow[r, "\epsilon/Z"] \arrow[dr, "{\epsilon,Z}"'] & \bullet \arrow[r, "{\epsilon,Z}"] & \bullet \arrow[r, "{a,Z}"] & q_a \\
 & & & q_\epsilon & &
\end{tikzcd}$$

After all these changes the set of states Q can be partitioned into Q_Σ^{PUSH} (the set of states in which pushing a symbol will require an input symbol) and $Q_\epsilon^{\mathrm{PUSH}}$ (from which a push will trigger an ϵ-transition). Analogously, $Q \times \Gamma$ can be divided into the disjoint subsets $(Q \times \Gamma)_\Sigma$ and $(Q \times \Gamma)_\epsilon$. Consequently, transitions of the new automaton can be divided into four groups:

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

Thus, weak pushdown automata have the same expressive power as pushdown automata. $\square$

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

$$\mathbf{exp} \to ((\mathbf{com}_3 \to \mathbf{com}_2) \to \mathbf{com}_1) \to \mathbf{com}_0$$

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 represented by a position of the following shape

$$\begin{array}{cccccccccc} r_0r_1 & r_2r_3 & r_2r_3 & r_2r_3 & d_3d_2 & d_3d_2 & r_2r_3 & d_3d_2 & d_3d_2 & d_1d_0 \\ \text{START} & \text{PUSH} & \text{PUSH} & \text{PUSH} & \text{POP} & \text{POP} & \text{PUSH} & \text{POP} & \text{POP} & \text{STOP} \end{array}$$

in which justification pointers are either to the preceding move or as shown below:

$$r_0\ r_1\ r_2\ r_3\ r_2\ r_3\ r_2\ r_3\ d_3\ d_2\ d_3\ d_2\ r_2\ r_3\ d_3\ d_2\ d_3\ d_2\ d_1\ d_0$$

We can think of r_0r_1 as an initialization sequence, d_1d_0 indicate that the automaton halted. Note that whenever O is to make a move he can play r_2 justified by the second move r_1 or, if there is an unanswered r_3, O can play d_3. However, by the bracketing condition that d_3 must always answer the last pending r_3, which corresponds exactly to the 'last-in first-out' principle. Therefore, at each turn O can start either a push sequence or a pop sequence (if there is a corresponding push not yet balanced with a pop) which corresponds exactly to the two kinds of actions performed by pushdown automata. Additionally, d_1 can be played 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. All such positions turn out to be the complete plays defined by $\lambda f.f(\lambda x.x)$. Their stack structure stems from the fact that the indeterminate procedure f may use $\lambda x.x$ in various ways, in particular the uses may be nested. The O-moves r_2 are then queries about $\lambda x.x$ in the sense of Example 3.4. In order to encode the transitions of a given automaton we have to control the order in which pop and push sequences can arise in a play, which will done by adorning the term with suitable side-effects in the spirit of Example 4.3.

Example 6.4. The term below, of type $((\mathbf{com} \to \mathbf{com}) \to \mathbf{com}) \to \mathbf{com}$, defines complete positions corresponding to a series of push actions followed by the same number of pops.

$$\lambda f.\mathbf{new}\ PUSH := \mathit{true}\ \mathbf{in}\ f(\lambda x.[!PUSH = \mathit{true}]; x; (PUSH := \mathit{false}))$$

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.

A similar term of type $\mathbf{exp} \to ((\mathbf{com} \to \mathbf{com}) \to \mathbf{com}) \to \mathbf{com}$ can be used to represent weak pushdown automata (the code for *pre* and *post* is shown in Figure 3):

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

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

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

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

Fig. 3. Code for *pre* and *post* used in the simulation of weak pushdown automata.

will be stored in a new copy of $STACK$. Note that while all r_2r_3 segments share 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, if so an input symbol is asked for when necessary and finally $STATE$ is 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. This corresponds to acceptance by final state with empty stack, as required.

Note that above we have used values of type **exp** to represent elements of Γ. If $|\Gamma| > |\Sigma|$ then elements of Γ will have to be encoded as tuples of values (this is possible as long as $N > 0$). Consequently, each context-free language is representable by a third-order IA term.

THEOREM 6.5. *$\mathcal{L}_3$ is the class of context-free languages.*

Remark 6.6. Our representation of pushdown automata uses three variables: $STATE$, $INPUT$ and $STACK$. However, the values of the last two never change after the first write, so they could actually be eliminated by branching on that value.

7. RECURSIVELY-ENUMERABLE LANGUAGES

In this section we demonstrate that any language accepted by a Turing machine can be represented by a fourth-order term.

It is well known that two stacks suffice to simulate a Turing machine but, although we can accommodate one, the bracketing condition prevents us from extending the representation scheme to two. Instead we will exploit the fact that the additional moves 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: 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 [Minsky 1967].

First we will introduce an auxiliary class of machines with Γ-stores (a nondeterministic generalization of queues) whose computations turn out representable by IA terms. In addition to stored elements, Γ-stores contain some bookkeeping information which can be used to verify if a Γ-store behaved like a queue. This check will turn out to possess a natural game-semantic reading, which will enable us to model proper queues as well.

Definition 7.1. A **Γ-store** stores characters from Γ. Its contents are described by a natural number n (the current size) and a function $f : \{0, \cdots, n\} \rightarrow \Gamma \times \{+, -\} \times \{+, -\}$ (the state of the Γ-store). The three fields of $f(i)$ will be referred to as

$$f(i).SYMBOL, f(i).ACCESSED \text{ and } f(i).MARKED.$$

The first one holds the stored character, the latter two indicate whether the ith entry has already been accessed or marked by the GET instruction. The initial state of a Γ-store is shown below:

	0
SYMBOL	!
ACCESSED	+
MARKED	−

i.e. $n = 0$ and $f(0) = (!, +, -)$, where ! is some (irrelevant) symbol from Γ (not considered to be stored but already deemed *accessed* and *unmarked*). The state of a Γ-store can be updated with two kinds of instructions:

—ADDc adds c to the store. The new state

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

is defined by $f \subseteq f'$, $f'(n+1) = (c, -, -)$.

—GET is the only access method which first chooses (if possible) two indices (i, j) such that $0 \leq j < i \leq n$ and

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

and then resets $f(i).ACCESSED, f(j).MARKED$ to $+$. That is to say, GET accesses an element not accessed thus far and marks another unmarked but already accessed element in front of the element being accessed. If no suitable pair (i, j) exists the operation fails. Otherwise, the result is the accessed symbol: $f(i).SYMBOL$.

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

	0	1	2	3	4	5
SYMBOL	!	a	b	c	b	a
ACCESSED	+	+	−	+	−	−
MARKED	+	+	−	−	−	−

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

	0	1	2	3	4	$\cdots$	n
SYMBOL	!	a	b	c	d	e	f
ACCESSED	+	+	+	+	−	−	−
MARKED	+	+	+	−	−	−	−

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 no future actions can modify later. For example, GET will never access b in the Γ-store from Example 7.2, because no suitable element will be available for marking. Thus, given the state of a Γ-store one can immediately tell whether it has behaved like a queue. Moreover, if all elements of a Γ-store have been accessed then the uses must have followed the queue discipline.

We incorporate Γ-stores in a simple machine model next.

Definition 7.4. A **Γ-machine** is a tuple $\langle Q, \Gamma, q_I, F, \delta^{\mathrm{ADD}}, \delta^{\mathrm{GET}} \rangle$ equipped with a Γ-store, where:

—$Q = Q^{\mathrm{ADD}} + Q^{\mathrm{GET}} + F$ is the set of states, $q_I \in Q^{\mathrm{ADD}}$ is the *initial* state, states in F are *final*;
—$\delta^{\mathrm{ADD}} : Q^{\mathrm{ADD}} \to Q \times \Gamma$ defines actions from states in Q^{ADD}: ADD $\pi_2(\delta^{\mathrm{ADD}}(q))$ is performed and the state changes to $\pi_1(\delta^{\mathrm{ADD}}(q))$;
—$\delta^{\mathrm{GET}} : Q^{\mathrm{GET}} \times \Gamma \to Q$ defines actions at states $q \in Q^{\mathrm{GET}}$: an attempt to execute GET takes place, if it succeeds returning $c \in \Gamma$ the next state is $\delta^{\mathrm{GET}}(q, c)$; otherwise the machine is stuck.

A Γ-machine makes nondeterministic transitions from the initial state until it reaches a final state. Then we say that it halts. Note that a failed attempt to perform a GET step does not count as halting (Q^{GET} and F are disjoint). Later on, in Definition 7.7, we will impose stricter acceptance conditions and define what it means for a Γ-machine to accept a word. However, for the time being, we focus on halting by final state.

To interpret runs of Γ-machines via positions we will extend the previously used encoding for pushdown automata. Recall that the positions corresponding to successful computations of pushdown automaton, e.g.

$$r_0 r_1 r_2 r_3 r_2 r_3 d_3 d_2 r_2 r_3 d_3 d_2 d_3 d_2 d_1 d_0,$$

were plays on $[\![((\mathbf{com}_3 \to \mathbf{com}_2) \to \mathbf{com}_1) \to \mathbf{com}_0]\!]$ generated by $\lambda f.f(\lambda x.x)$. $r_2 r_3$ and $d_3 d_2$ indicated pushing and popping respectively. If we work with the fourth-order type $(((\mathbf{com}_4 \to \mathbf{com}_3) \to \mathbf{com}_2) \to \mathbf{com}_1) \to \mathbf{com}_0$ these stack traces will be in 1-1 correspondence with plays from $\mathsf{comp}([\![\lambda h.h(\lambda f.f\,\Omega)]\!])$. Note that the occurrence of Ω prevents P from replying after O uses the previously unavailable r_4. Consequently, the term defines no complete play in which r_4 is used. Of course, in our encoding Ω will be replaced with a different term. The additional move r_4 will turn 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 GET.

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. For technical convenience we will assume that the initial state of the Γ-store results from a special ADD (executed once at the very beginning), which introduces the unmarked accessed symbol ! but does not affect the state. Let us consider the sequence representing n consecutive ADD instructions (plus the initializing one)

$$\underset{\text{START}}{r_0}\ r_1\ \underset{\text{INIT}}{r_2\ r_3}\ \underset{\text{ADD1}}{r_2\ r_3}\ \cdots\ r_2\ r_3\ \cdots\ r_2\ r_3\ \cdots\ \underset{\text{ADDn}}{r_2\ r_3}$$

Above and in what follows pointers that would point to the preceding moves have been omitted.

The interpretation of GET 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 $r_4 r_3$

$$r_0\ r_1\ \underset{\text{INIT}}{r_2\ r_3}\ \underset{\text{ADD1}}{r_2\ r_3}\ \cdots\ r_2\ r_3\ \cdots\ \underset{\text{ADDi}}{r_2\ r_3}\ \cdots\ \underset{\text{ADDn}}{r_2\ r_3}\ r_4\ r_3$$

This behaviour is consistent with the strategy interpreting $\lambda h.h(\lambda f.f(f\,\Omega))$ (we have $r_0 r_1 r_2 r_3 r_4 r_3 \in [\![\lambda h.h(\lambda f.f(f\,\Omega))]\!]$). Note that when this strategy is followed then, due to Ω, P will not respond 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 (in his view) the blocks $r_2 r_3$ corresponding to ADD operations with index $j < i$; marking will correspond to playing $r_4 d_4$, where r_4 is justified by r_3 from the jth block:

$$r_0\ r_1\ \underset{\text{INIT}}{r_2\ r_3}\ \underset{\text{ADD1}}{r_2\ r_3}\ \cdots\ \underset{\text{ADDj}}{r_2\ r_3}\ \cdots\ \underset{\text{ADDi}}{r_2\ r_3}\ \cdots\ \underset{\text{ADDn}}{r_2\ r_3}\ r_4\ r_3\ r_4\ d_4$$

This time the last P-move coincides with the response defined by $\lambda h.h(\lambda f.f\ \mathit{skip})$ ($r_0 r_1 r_2 r_3 r_4 d_4 \in [\![\lambda h.h(\lambda f.f\ \mathit{skip})]\!]$).

—Finally, we would like the players to answer the questions raised in the first stage with $d_3 d_4$:

$$r_0\ r_1\ \underset{\text{INIT}}{r_2\ r_3}\ \underset{\text{ADD1}}{r_2\ r_3}\ \cdots\ \underset{\text{ADDj}}{r_2\ r_3}\ \cdots\ \underset{\text{ADDi}}{r_2\ r_3}\ \cdots\ \underset{\text{ADDn}}{r_2\ r_3}\ r_4\ r_3\ r_4\ d_4\ d_3\ d_4$$

Again, this is consistent with $\lambda h.h(\lambda f.f(f\Omega))$, as $[\![\lambda h.h(\lambda f.f(f\Omega))]\!]$ contains $r_0 r_1 r_2 r_3 r_4 r_3 d_3 d_4$.

Observe that the moves played in the third stage will hide moves introduced in the previous two steps from O-view. Thus, when the simulation of a GET step is

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

Fig. 4. *add* and *get*

finished, the O-view will consist of the old O-view (before the first stage) and a segment $r_4 d_4$. Consequently, subsequent ADD and GET steps can be modelled in the same way as defined above and the corresponding justified sequences will always be legal positions. Moreover, the O-view of each sequence representing a series of ADD's and GET's will consist of $r_0 r_1$ and all segments $r_2 r_3$ corresponding to all ADD actions intertwined with segments $r_4 d_4$. When the machine reaches the final state, we would like the players to complete the play with a series of $d_3 d_2$ segments (which will answer the questions played in *ADD* blocks) and, finally, $d_1 d_0$ so that a complete play is reached.

Example 7.5. The position below represents actions of the Γ-store from Example 7.2.

$$
\underbrace{r_0\ r_1}_{\text{START}}\ \underbrace{r_2\ r_3}_{\text{INIT}}\ \underbrace{r_2\ r_3}_{\text{ADD}}\ \underbrace{r_2\ r_3}_{\text{ADD}}\ \underbrace{r_2\ r_3}_{\text{ADD}}\ \underbrace{r_4\ r_3\ r_4\ d_4\ d_3\ d_4}_{\text{GET}}\ \underbrace{r_2\ r_3}_{\text{ADD}}\ \underbrace{r_4\ r_3\ r_4\ d_4\ d_3\ d_4}_{\text{GET}}\ \underbrace{r_2\ r_3}_{\text{ADD}}
$$

We have already seen that the term corresponding to a given Γ-machine will have to combine the behaviour of $\lambda h.h(\lambda f.f\ \mathit{skip})$ and $\lambda h.h(\lambda f.f(f\,\Omega))$ and "switch" between them suitably. As before we will use the variable *STATE* to store the state of the machine after each step is simulated. We will make sure that $!\mathit{STATE}$ after r_3 (for ADD) and after the second d_4 (for GET) is always equal to the state of the machine after it executes all the represented steps. In addition, other auxiliary variables (called *ACCESS*, *MARKING*) will be used to trace the three stages involved in simulating GET, whereas *FIRST* will guide the initial phase of the simulation when the Γ-store has to be initialized in a fake ADD step.

We define the term as

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

where the *add* and *get* subroutines are given in Figure 4. In particular *get* is responsible for switching between $f\,\Omega$ and *skip*.

Elements of the Γ-store are represented by the variables SYMBOL,$\mathit{ACCESSED}$ and MARKED in much the same way as the contents of the stack were represented before, i.e. each block $r_2 r_3$ will have its local copies of the variables. In contrast, STATE, FIRST, ACCESS, $\mathit{MARKING}$ are defined "globally". If $|\Gamma| > |\Sigma|$, tuples of variables instead of single variables will have to be used to store symbols from Γ.

THEOREM 7.6. *Let M be the term defined above. Then* $\mathsf{comp}([\![M]\!])$ *contains exactly the positions representing successful computations of the associated* Γ*-machine.*

PROOF. We use induction on the length of positions to show that each (even-length) position $s \in [\![M]\!]$ can be extended to a complete one if and only if the next moves follow our representation scheme for the given Γ-machine. In particular we show that whenever O deviates from our convention or tries to model an illegal step, e.g. a GET step when $!\mathit{STATE} \notin Q^{\mathrm{GET}}$, P will not be able to reply and thus the undesirable O-moves will never lead to complete positions.

For a start, we have $r_0 r_1 \in [\![M]\!]$ and this is the only play of length 2 in $[\![M]\!]$. Note that $!\mathit{STATE}$ at r_1 is equal to the initial state $q_I \in Q^{\mathrm{ADD}}$ and $!\mathit{FIRST} = \mathit{true}$.

Let us consider a position $s \in [\![M]\!]$ which represents a number of transition steps (ADD or GET). We consider all possible ways of extending it in turn.

—Suppose $!\mathit{STATE} \in Q^{\mathrm{ADD}}$ at the end of s. Then r_2 will always trigger r_3 (corresponding to the outer occurrence of f). Note that hidden moves interpreting $[!\mathit{STATE} \in Q^{\mathrm{ADD}}]; \mathit{add}$ will precede r_3.
  - If $!\mathit{FIRST}$ is *true* then the associated SYMBOL and $\mathit{ACCESSED}$ variables will be set to ! and + respectively, and FIRST will be changed to *false* so that this case never happens again.
  - If $!\mathit{FIRST}$ is *false* then STATE and SYMBOL will be updated according to δ^{ADD}, thus modelling an ADD step of the machine.

  If $!\mathit{STATE} \in Q^{\mathrm{ADD}}$ and O plays d_3 or d_1 (if legal) rather than r_2, P will have no reply because of the two occurrences of the assertion $[!\mathit{STATE} \in F]$. Similarly, $[!\mathit{STATE} \in Q^{\mathrm{GET}}]$ will prevent P from replying if O plays r_4.

  Hence, if $!\mathit{STATE} \in Q^{\mathrm{ADD}}$ at the end of s, the only moves possibly leading to a complete play must represent an ADD operation. Additionally, immediately after $r_0 r_1$ only the auxiliary ADD initializing the Γ-store can take place and in both cases $!STATE$ changes in the same way as the state of the machine.

—Suppose $!STATE \in Q^{\mathrm{GET}}$ at the end of s. Then if any of r_2, d_1, d_3 (if legal) is played, P will not reply because of one of the assertions. So, only r_4 can be used to complete the current position. The move can be justified by any visible occurrence of r_3 and our convention ensures that precisely the r_3 moves used to model previous ADD operations are visible at that point. Thus the choice of the justifier corresponds to the choice of $0 \leq i \leq n$. r_4 is a query about the argument to the outermost f, so P's reply is determined by $[!STATE \in Q^{\mathrm{GET}}]; get$. Because $!ACCESS = \mathit{false}$ the first branch will be chosen in get. Then $!ACCESSED$ in the ith ADD block is examined: $[!ACCESSED = -]$ makes sure that P will have no reply if the ith element in the Q-store has already been accessed. Thus the play can proceed only if $!ACCESSED = -$; $ACCESS$ will then be set to true and r_3 justified by r_2 from the ith block will be played (corresponding to the inner occurrence of f).
After that, since $STATE$ remains unchanged, the next O-move must be either r_4 or d_3. The latter possibility would finish the play because of $[!MARKING = \mathit{true}]$, so if a complete position is to be reached O has to play r_4 again. This time (by visibility) the justifier can be any r_3 from an ADD block to the left of the ith block or the r_3 just played by P. The latter would be a query about the argument to the inner f and, as this is Ω, P would not reply. Thus, the only way forward is to follow the representation scheme and use an r_3 move from a visible ADD block as a justifier for r_4 (this corresponds to the choice of j such that $0 \leq j < i$). As before such an r_4 move is a query about the argument to the outer f but, because $!ACCESS$ equals true now, the second branch of get will be played out, i.e. the hidden play will verify whether $!ACCESSED$ and $!MARKED$ in the jth block are equal to $+$ and $-$ respectively and whether $!MARKING$ is false. Subsequently, $MARKING$ will be set to true for the sake of future moves, $MARKED$ in the jth block will be changed to $+$ and d_4 will be played in response to r_4. At this point we can be certain that an appropriate (i, j) pair has been found, so a GET step can take place. The state has not changed yet (we still have $!STATE \in Q^{\mathrm{GET}}$), so once again in order to complete the position O can play either r_4 or d_3. This time the former is not a good idea, either because of Ω or $[!MARKING = \mathit{false}]$. But we can make progress after d_3 because $!MARKING = \mathit{true}$. This will lead to the interpretation of the rest of the first branch of get, i.e. $ACCESS$ and $MARKING$ will be reset to their initial values, the field $ACCESSED$ in the ith ADD block will be set to $+$ and $STATE$ will be changed using the $SYMBOL$ stored in the ith block. In this way the players reach the end of get, so P will play d_4 (completing the interpretation of the argument to the outer occurrence of f).
Hence, if $!STATE \in Q^{\mathrm{GET}}$ the only way to complete a position is to interpret a GET step (if possible).

—Finally, assume that $!STATE \in F$ at the end of s. Then P will not be able to respond to r_2 or r_4 (because of $[!STATE \in Q^{\mathrm{ADD}}]$ and $[!STATE \in Q^{\mathrm{GET}}]$). However, d_3 will no longer be blocked (P will reply with d_2) and when it is legal to play d_1 P will duly finish with d_0, as required.

Consequently, the complete positions generated by M are exactly the representations of successful computations of the given Γ-machine. □

In order to relate Γ-machines to Turing machines we strengthen the acceptance conditions.

Definition 7.7. Let $\Sigma \subseteq \Gamma$ be an input alphabet. Then, we say that a Γ-machine $\langle Q, \Gamma, q_I, F, \delta^{\mathrm{ADD}}, \delta^{\mathrm{GET}} \rangle$ *accepts* $x_1 \cdots x_k \in \Sigma^*$ if it halts when the initial state of its Γ-store is

	0	1	$\cdots\cdots$	k
SYMBOL	!	x_1	$\cdots\cdots$	x_k
ACCESSED	+	−	$\cdots\cdots$	−
MARKED	−	−	$\cdots\cdots$	−

Moreover, we require that all elements of the Γ-store must have been accessed when the machine halts. A Γ-machine accepts a language $L \subseteq \Sigma^*$ if it accepts precisely the words in L.

Γ machines with the above notion of acceptance can be shown to be universal. As already pointed out in Example 7.3, they will accept a word only when the Γ-store behaves like a queue which is emptied at the end. Therefore, Γ-machines behave exactly like finite machines equipped with a queue. Simple machines of that sort were first considered in the form of tag systems [Post 1943] and proved Turing-strong in [Minsky 1967], so the universality of Γ-machines is not surprising. Γ-machines are actually more flexible than tag systems and the proof of their universality is quite routine.

THEOREM 7.8. *For any Turing machine there exists an equivalent* Γ*-machine such that* $\Gamma = \Gamma' \cup \{|, h\}$*, where* Γ' *is the tape alphabet of the Turing machine.*

PROOF. The basic idea is to use the queue to store the tape of the Turing machine delimited with | on the left and on the right. The special letter h will mark the position of the head (the symbol to the left of h can then be taken to correspond to the one under the head). Thus, before the simulation starts the input string has to be framed in |'s and h has to be inserted so that instead of the input string $x_1 \ldots x_k$ the queue contains $|x_1 \ldots x_k h|$. This, as well as modifications corresponding to transitions of the Turing machine, can be modelled by a series of dequeuing and enqueuing operations in which the representation of the tape is processed in a circular fashion. When the machine reaches its final state, the Γ-machine must start emptying the queue and reach its final state after processing the last |. □

Recall that it is undecidable whether a Turing machine starting from the empty tape halts. By the previous Theorem, it undecidable whether a Γ-machine accepts the empty word ϵ.

PROPOSITION 7.9. *For any* Γ*-machine there exists a finitary term*

$$M : (((\mathbf{com} \to \mathbf{com}) \to \mathbf{com}) \to \mathbf{com}) \to \mathbf{com}$$

such that the machine accepts ϵ *if and only if* $\mathsf{comp}(\llbracket M \rrbracket) \neq \emptyset$.

PROOF. We use the construction from Theorem 7.6, which yields a term with the requisite type whose complete plays correspond to halting computations of the

Γ-machine. However, this time we are only interested in the positions in which the Γ-store behaves like a queue which becomes empty at the end. This is equivalent to stipulating that all $ACCESSED$ fields of the store contain $+$ at the end. Fortunately, this condition can be enforced through a simple modification of the term.

Recall that each complete position corresponding a halting run of a Γ-machine ends in a series of $d_3 d_2$ segments followed by the final moves $d_1 d_0$. The order in which the answers $d_3 d_2$ are played can be viewed as a right-to-left scan of the Γ-store, which we can use to verify that $!ACCESSED = +$ for every element of the store. This can be achieved by replacing

$$f\,([!STATE \in Q^{\mathrm{GET}}]; get);$$

in the term from Theorem 7.6 with

$$(f\,([!STATE \in Q^{\mathrm{GET}}]; get)); [!ACCESSED = +];$$

□

Theorem 7.10. *Observational equivalence in finitary* IA *is undecidable.*

Proof. By the preceding Proposition it is undecidable whether $\mathsf{comp}([\![M]\!]) = \emptyset$ for $M : (((\mathbf{com} \to \mathbf{com}) \to \mathbf{com}) \to \mathbf{com}) \to \mathbf{com}$. By Theorem 3.6 $\mathsf{comp}([\![M]\!]) = \emptyset$ if and only if $M \cong \lambda h.\Omega$, so observational equivalence of terms of type $(((\mathbf{com} \to \mathbf{com}) \to \mathbf{com}) \to \mathbf{com}) \to \mathbf{com}$ is not decidable. □

The divergent term $\Omega : \mathbf{com}$ has played an important role in our encodings where it was used to eliminate complete positions in which O failed to cooperate in implementing the representation scheme. Without Ω we would have to find a way of dealing with complete positions that do not correspond to any machine runs. Fortunately, it is possible to modify the terms used previously so that the undesirable complete positions are distinguished from the good ones, for example, 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 $\cdots \to \mathbf{exp}$ rather than $\cdots \to \mathbf{com}$. To this end we can introduce a global variable OK (initialized to *true*), replace each occurrence of $\Omega : \mathbf{com}$ by $OK := \mathit{false}$ and arrange for P to play $!OK$ as the final move. Then the successful runs correspond to complete plays ending in *true*. Moreover, the problem whether any such run exists can be stated as an equivalence query involving the term just described and its copy in which $!OK$ is replaced with *false*. Thus the undecidability result also applies to Ω-free terms of finitary IA.

To represent any recursively enumerable language, we need to make some provision for inputing symbols so that the Γ-store can be initialized with any string from Σ^* before the machine tries to accept it. To achieve this we will modify the term from Proposition 7.9. 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 $r_2 r_3$ in the same way as ADD actions, but $STATE$ will remain unchanged throughout (and equal to q_I). We must also 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

justified by the r_3 in the first r_2r_3 segment (which introduces the dummy accessed element !). This will set a global variable (named *INPUT*), whose initial value is *true*, to *false*. Once this happens, the simulation proper will commence in the old way.

THEOREM 7.11. *$\mathcal{L}_i$ $(i \geq 4)$ is the class of recursively enumerable languages.*

PROOF. Consider the term from Proposition 7.9 and add $\lambda c.$ at the front to form a term of type $\mathbf{exp} \to (((\mathbf{com} \to \mathbf{com}) \to \mathbf{com}) \to \mathbf{com}) \to \mathbf{com}$. Add the global initialization $\mathit{INPUT} := \mathit{true}$ and replace $(\mathit{STATE}, \mathit{SYMBOL}) := \delta^{\mathrm{ADD}}(!\mathit{STATE})$ in *add* with

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

to enable the input stage. Further, in order to allow O to finish the input phase by r_4 justified by the first r_3, replace $[!\mathit{STATE} \in Q^{\mathrm{GET}}]; \mathit{get}$ with

$$\begin{array}{l}\mathbf{if}\ !\mathit{INPUT} = \mathit{true}\ \mathbf{then}\ \ [!\mathit{ACCESSED} = +]; (\mathit{INPUT} := \mathit{false}) \\ \qquad\qquad\qquad\qquad\quad \mathbf{else}\ \ ([!\mathit{STATE} \in Q^{\mathrm{GET}}]; \mathit{get})\end{array}$$

Note that no complete position will be reached if O plays r_4 which is not justified by the first r_3 because of the assertion $[!\mathit{ACCESSED} = +]$. After the above modifications, by Theorem 7.8 and Proposition 7.9, we get a term representing any given recursively enumerable language.

Conversely, any language representable by a finitary IA term (indeed any IA term) is recursively enumerable. This is because all the basic strategies defining the game model of IA are recursive and composition is effective in the following sense: given $s \in P^{\circledast}_{A \Rightarrow B}$, $t \in P_{B \Rightarrow C}$, $v \in P_{A \Rightarrow C}$ and $u \in \mathit{inter}(A, B, C)$ it is decidable whether $u \upharpoonright A, B = s$, $u \upharpoonright B, C = t$ and $u \upharpoonright A, C = v$. Note that for a given term $M : T$ there is a bound on the number of compositions required to produce any $s \in [\![T]\!]$ (roughly the number of nodes in M's syntactic tree). Thus, given $M : T$ one can construct a Turing machine which generates positions on $[\![T]\!]$ along with a proof that they are in $[\![M]\!]$ in the form of potential interaction sequences. By previous remarks the correctness of that proof will be decidable. Since it is also decidable whether a position represents a given $x \in \Sigma^*$, for any language L representable by an IA term we can construct a Turing machine which, given $x \in \Sigma^*$, will halt if and only if $x \in L$. ☐

8. CONCLUSION

We have proved a new result about a programming language using its semantics. This is probably a value in itself. After all, one motivation for having models is the expectation that they can provide original insights about the associated languages.

The undecidability result for IA can be adapted to the call-by-value setting by relating the associated game models, as outlined in [Murawski 2003]. Despite some fundamental differences between the games involved in modelling call-by-name and call-by-value, for each of the IA terms considered in our representation theorems, it is possible to construct a call-by-value term inducing the same semantics. As a result, program equivalence in call-by-value IA can be shown undecidable at fifth order.

One would hope that our techniques can also be used to shed a new light on program equivalence in PCF [Loader 1997]. However, the encodings presented in the paper rely on the availability of state too much and in order to become relevant to PCF they would have to be reworked considerably.

In any case, what we show is the potential of game semantics to provide geometric intuitions about computation which seem difficult to notice otherwise.

REFERENCES

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

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

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

Abramsky, S. and McCusker, G. 1997b. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. In *Algol-like languages*, P. W. O'Hearn and R. D. Tennent, Eds. Birkhaüser, 297–329.

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

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

Constable, R. L. and Muchnick, S. S. 1972. Subrecursive program schemata I & II: I. Undecidable equivalence problems; II. Decidable equivalence problems. *JCSS 6(6)*, 480–537.

Ghica, D. R. and McCusker, G. 2000. Reasoning about Idealized Algol using regular expressions. In *Proceedings of ICALP*. Lecture Notes in Computer Science, vol. 1853. Springer-Verlag, 103–115.

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

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

Hyland, J. M. E. and Ong, C.-H. L. 2000. 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.

Ibarra, O. H., Su, J., Dang, Z., Bultan, T., and Kemmerer, R. 2000. Counter machines: Decidable properties and applications to verification problems. In *Proceedings of MFCS*. Lecture Notes in Computer Science, vol. 1893. Springer-Verlag, 426–435.

Laird, J. 1998. A semantic analysis of control. Ph.D. thesis, University of Edinburgh.

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

McCusker, G. 2001. On the semantics of Idealized Algol without the bad-variable constructor. Technical Report, SOCS, University of Sussex.

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

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

Nickau, H. 1996. Hereditarily sequential functionals: A game-theoretic approach to sequentiality. Ph.D. thesis, Universität-Gesamthochschule-Siegen.

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

Post, E. 1943. Formal reductions of the combinatorial decision problem. *American Journal of Mathematics 65*, 196–215.

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

Sénizergues, G. 2001. L(A)=L(B)? decidability results from complete formal systems. *Theoretical Computer Science 251(1-2)*, 1–166.

Sipser, M. 1997. *Introduction to the Theory of Computation*. PWS Publishing Company.