IPSJ Transactions on Programming Vol.5 No.4 17–26 (Sep. 2012)

Regular Paper

May & Must-Equivalence of Shared Variable Parallel Programs in Game Semantics

Keisuke Watanabe[1,†1] Susumu Nishimura[1,a)]

Received: February 13, 2012, Accepted: April 26, 2012

Abstract: We present a game semantics for an Algol-like language with shared variable parallelism. On contrary to deterministic sequential programs, whose semantics can be characterized by observing termination behaviors, it is crucial for parallel programs to observe not only termination but also divergence, because of nondeterministic scheduling of parallel processes. In order to give a more appropriate foundation for modeling parallelism, we base our development on Harmer's game semantics, which concerns not only may-convergence but also must-convergence for a nondeterministic programming language EIA. The game semantics for the Algol-like parallel language is shown to be fully abstract, which indicates that the parallel command of our Algol-like language adds no extra power than nondeterminism provided by EIA. We also sketch how the equivalence of two parallel programs can be reasoned about based on the game semantical interpretation.

Keywords: shared-variable parallelism, may & must-convergence, full abstraction, game semantics

1. Introduction

Programming multiprocessors is a much more challenging task than programming uniprocessors. Parallel programs running on commodity shared memory multiprocessors are inherently nondeterministic and their access to memory is asynchronous due to preemptive scheduling. It is a central concern of parallel programming how one can rightly keep the integrity of shared resources against the asynchrony [1].

In this paper, we propose a game semantics for shared variable parallel programs. Game semantics has been successfully applied to modeling deterministic languages such as PCF and Idealized Algol (IA) [2], [3], [4]. In game semantics, program execution is modeled by a game consisting of two participants, called *opponent* and *player*, representing the environment and the program, respectively. The semantics of a given program is specified by a game strategy, a collection of possible alternating opponent/player moves.

There are some studies on the game theoretical analysis for shared memory parallel programs [5], [6], but they are devoted to the so-called angelic parallelism based on *may-equivalence*. Two programs are regarded as may-equivalent if one program may-converges (i.e., it has a choice of terminating evaluation path) then so does the other, in any context.

May-equivalence is unsatisfactory for the purpose of modeling shared variable parallel programs, however. For instance, consider the following parallel program.

$$\mathsf{parallel}\ b=0, f_0=0, f_1=0, v_0=0, v_1=0\ \mathsf{in}$$
$$\left\{\begin{array}{l} f_0 := 1;\ \mathsf{while}\ !f_1 \neq 0\ \mathsf{do\ skip}; \\ v_0 := !b; b := 1-!b; f_0 := 0 \end{array}\right\}$$
$$\Big\|\ \left\{\begin{array}{l} f_1 := 1;\ \mathsf{while}\ !f_0 \neq 0\ \mathsf{do\ skip}; \\ v_1 := !b; b := 1-!b; f_1 := 0 \end{array}\right\}$$

This program executes two parallel processes (delimited by ||) that share common local variables b, f_0, f_1, v_0, v_1. It is intended to set the variables v_0 and v_1 either 0 or 1 exclusively, using the variable b for a shared binary counter. Each parallel process *spin locks* over f_0 or f_1 in order to acquire the shared resource b. We can see that, whenever the program gracefully terminates (e.g., when the entire execution of one parallel process precedes the other's), we will have the intended result. Thus this parallel program is may-equivalent to a program that nondeterministically executes either $v_0 := 0; v_1 := 1$ or $v_1 := 0; v_0 := 1$.

However, there are many other possibilities of interleaving that cause divergence. For instance, the program does not converge with the following infinite sequence:

$$f_0 := 1,\ \underline{f_1 := 1},\ \underline{!f_0 = 1},\ !f_1 = 1,\ \underline{!f_0 = 1},\ \underline{!f_0 = 1}, \ldots,$$

which repeats $!f_1 = 1$ (reading value 1 from f_1) and $\underline{!f_0 = 1}$ alternatingly. (The underlines are put to distinguish different execution threads.)

This explains why may-equivalence is not satisfactory: May-equivalence can only guarantee two may-equivalent parallel programs to agree when they both happen to terminate, but it does not exclude the possibility that one terminates whereas the other diverges, due to nondeterminism caused by asynchronous parallel execution.

In this paper, we present a game semantics for a parallel programming language, for which we define contextual equivalence

[1] Department of Mathematics, Faculty of Science, Kyoto University, Kyoto 606–8502, Japan
[†1] Presently with Tezukayama Junior and Senior High School
[a)] susumu@math.kyoto-u.ac.jp

by means of *may&must-equivalence*. Two programs are regarded as equal if they are may-equivalent and as well one's guarantee of termination implies the other's.

For this, we base our development on the game semantic framework for EIA, an extension of IA with erratic finite nondeterminism [7], [8]. We give a game semantics for a programming language that further supports shared variable parallelism and show the full abstraction result. Interestingly, this result indicates that the parallel construct adds no extra expressive power other than provided by the erratic nondeterminism. We also briefly discuss how the contextual (in)equivalence for a certain fragment of the programming language can be argued on regular language models, in much the same way as done for a fragment of IA [9].

1.1 Related Work

Process calculi [10], [11] have been successful in describing and analyzing the complex behavior of the systems of concurrent processes that communicate via (perhaps asynchronous) message passing over channels. A major concern in the studies of those calculi is whether the given systems of concurrent processes exhibit the same behavior, even if they are supposed to be nonterminating. In contrast to this, in the present paper we are concerned with semantical analysis of parallel protocols (e.g., those for mutual exclusion) which are implemented by means of asynchronous shared variables. We would like to argue if a given implementation of a parallel protocol successfully gives the intended result on termination. For this, we will argue the contextual equality of programs by observing the (may&must) termination.

There are some preceding studies on the semantics of shared variable parallel programs. Brookes has given a fully abstract semantics based on transition traces [12]. Ghica, Murawski, et al. [5], [6] have given a fully abstract game model for an Algol-like shared variable parallel programming language. These only concern may-equivalence of parallel programs, however. To our knowledge, there have been no studies of game theoretical analysis that argue may&must-equivalence properties of shared variable parallel programs.

The game semantics given in Refs. [5], [6] is quite different from ours in modeling parallel interleaving. Ours conservatively extends Harmer's game semantics for nondeterminism [8], in which opponent (resp., player) makes moves at odd (resp., even) positions in each game play, whereas theirs abandons this parity constraint in order to allow more flexible interleaving. Each of the two approaches has its own merits and demerits. It seems difficult to extend their game semantics to include divergence property as done in Harmer's game semantics, in which the parity constraint provides a sharp distinction between even-length terminating game plays and odd-length diverging ones. On the other hand, our parallel extension requires a technical refinement and a syntactic constraint to the parallel programming language, in order to achieve appropriate modeling of interleaving (Section 4.2). We hope that our proposal gives a different perspective on the topic and that it will give rise to a compromised solution in a future.

Outline. The rest of the paper is structured as follows. Section 2 defines the syntax and the operational semantics of an Algol-like programming language with parallel extension. Giving the notational and conceptual backgrounds for Harmer's game model for nondeterminism [8] in Section 3, we develop a game semantics for parallelism and show the full abstraction result in Section 4. Section 5 discusses how contextual (in)equalities of parallel programs can be checked in the game semantic model. Finally Section 6 concludes the paper.

2. An Algol-like Language with Parallelism

We define an Algol-like language, called $\mathsf{EIA}_{\mathsf{par}}$, for shared variable parallel programming. This extends EIA [7], [8], a nondeterministic variant of Idealized Algol, with a few parallel constructs.

The types of the language $\mathsf{EIA}_{\mathsf{par}}$ are defined as:

$$\mathtt{T} ::= \mathtt{nat} \mid \mathtt{com} \mid \mathtt{var} \mid \mathtt{T} \to \mathtt{T},$$

where `nat`, `com`, and `var` are base types representing natural numbers, commands, and mutable variables, respectively.

The syntax of the terms of the language is defined by the following grammar.

$$\begin{aligned} M ::= {} & x \mid n \mid M \star M \mid \lambda x^{\mathtt{T}}.M \mid MM \mid \mathsf{fix}_{\mathtt{T}}\, M \mid \mathsf{skip} \\ & \mid \mathsf{seq}_{\mathtt{T}}\, M\, M \mid \mathsf{ifz}_{\mathtt{T}}\, M \,\mathsf{then}\, M \,\mathsf{else}\, M \mid \mathsf{assign}\, M\, M \\ & \mid \mathsf{deref}\, M \mid \mathsf{cas}\, M\, M\, M \mid \mathsf{mkvar}\, M\, M\, M \\ & \mid \mathsf{newvar}\, x = n \,\mathsf{in}\, M \mid M \,\mathsf{or}\, M \\ & \mid \mathsf{parallel}\, x_1 = k_1, \ldots, x_m = k_m \,\mathsf{in}\, M \| M \,\mathsf{finally}\, M \ (m \geq 0) \end{aligned}$$

The terms are either PCF terms (natural numbers, λ-terms, and general recursion), Algol terms (commands including those for mutable variables), erratic nondeterminism (or), or parallel constructs. We write $\star$ to stand for binary operations on natural numbers.

We write a *type judgment* as $\Gamma \vdash M : \mathtt{T}$, where Γ is a *typing context*, a finite mapping from identifiers to types. In particular, a typing context Γ is called a `var`-*context* if every identifier in Γ is assigned the type `var`. The typing rules for $\mathsf{EIA}_{\mathsf{par}}$ are given in **Fig. 1**.

In what follows, we will occasionally write $!M$ for $\mathsf{deref}\ M$ and $M := N$ for $\mathsf{assign}\ M\ N$. Also, some type annotations may be omitted unless ambiguity can arise.

The operational semantics for $\mathsf{EIA}_{\mathsf{par}}$ is defined in the style of small-step operational semantics. We define *canonical forms*, ranged over by V, as a subset of terms:

$$V ::= n \mid \lambda x^{\mathtt{T}}.M \mid \mathsf{skip} \mid v \mid \mathsf{mkvar}\, M\, N\, L,$$

where v ranges over identifiers of type `var`.

A *store* s is a mapping from identifiers of type `var` to natural numbers. We write $\langle M, s\rangle \longrightarrow \langle M', s'\rangle$ for the *1-step reduction*, which corresponds to a single atomic computation that updates s to s' as the side effect. We define the 1-step reduction relation as in **Fig. 2**. Every reduction is defined relative to *evaluation context* E, where E is a term containing a single hole $[\,]$. We write $E[M]$ for the term obtained by filling the hole in E with term M.

We write $\langle M, s\rangle \longrightarrow^* \langle M', s'\rangle$ for the reflexive transitive closure of 1-step reductions. We say the evaluation of a program

$$\frac{}{\Gamma, x : \mathtt{T}, \Gamma' \vdash x : \mathtt{T}} \qquad \frac{}{\Gamma \vdash n : \mathtt{nat}} \qquad \frac{\Gamma \vdash M : \mathtt{nat} \quad \Gamma \vdash N : \mathtt{nat}}{\Gamma \vdash M \star N : \mathtt{nat}}$$

$$\frac{\Gamma, x : \mathtt{T}_1 \vdash M : \mathtt{T}_2}{\Gamma \vdash \lambda x^{\mathtt{T}_1}.M : \mathtt{T}_1 \to \mathtt{T}_2} \qquad \frac{\Gamma \vdash M_1 : \mathtt{T}_1 \to \mathtt{T}_2 \quad \Gamma \vdash M_2 : \mathtt{T}_1}{\Gamma \vdash M_1 M_2 : \mathtt{T}_2} \qquad \frac{\Gamma \vdash M : \mathtt{T} \to \mathtt{T}}{\Gamma \vdash \mathsf{fix}_{\mathtt{T}}\ M : \mathtt{T}}$$

$$\frac{}{\Gamma \vdash \mathsf{skip} : \mathtt{com}} \qquad \frac{\Gamma \vdash M_1 : \mathtt{com} \quad \Gamma \vdash M_2 : \mathtt{T} \quad \mathtt{T} \in \{\mathtt{nat}, \mathtt{com}\}}{\Gamma \vdash \mathsf{seq}_{\mathtt{T}}\ M_1\ M_2 : \mathtt{T}}$$

$$\frac{\Gamma \vdash M : \mathtt{nat} \quad \Gamma \vdash M_1 : \mathtt{T} \quad \Gamma \vdash M_2 : \mathtt{T} \quad \mathtt{T} \in \{\mathtt{nat}, \mathtt{com}\}}{\Gamma \vdash \mathsf{ifz}_{\mathtt{T}}\ M\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2 : \mathtt{T}} \qquad \frac{\Gamma \vdash M : \mathtt{var} \quad \Gamma \vdash N : \mathtt{nat}}{\Gamma \vdash \mathsf{assign}\ M\ N : \mathtt{com}}$$

$$\frac{\Gamma \vdash M : \mathtt{var}}{\Gamma \vdash \mathsf{deref}\ M : \mathtt{nat}} \qquad \frac{\Gamma \vdash L : \mathtt{var} \quad \Gamma \vdash M : \mathtt{nat} \quad \Gamma \vdash N : \mathtt{nat}}{\Gamma \vdash \mathsf{cas}\ L\ M\ N : \mathtt{nat}}$$

$$\frac{\Gamma \vdash M : \mathtt{nat} \to \mathtt{com} \quad \Gamma \vdash N : \mathtt{nat} \quad \Gamma \vdash L : \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}}{\Gamma \vdash \mathsf{mkvar}\ M\ N\ L : \mathtt{var}}$$

$$\frac{\Gamma, x : \mathtt{var} \vdash M : \mathtt{com}}{\Gamma \vdash \mathsf{newvar}\ x = n\ \mathsf{in}\ M : \mathtt{com}} \qquad \frac{\Gamma \vdash M_1 : \mathtt{nat} \quad \Gamma \vdash M_2 : \mathtt{nat}}{\Gamma \vdash M_1\ \mathsf{or}\ M_2 : \mathtt{nat}}$$

$$\frac{x_1 : \mathtt{var}, \ldots, x_m : \mathtt{var} \vdash P_i : \mathtt{com}\ (i = 1, 2) \qquad \Gamma, x_1 : \mathtt{var}, \ldots, x_m : \mathtt{var} \vdash Q : \mathtt{com}}{\Gamma \vdash \mathsf{parallel}\ x_1 = k_1, \ldots, x_m = k_m\ \mathsf{in}\ P_1 \| P_2\ \mathsf{finally}\ Q : \mathtt{com}}$$

Fig. 1 Typing rules.

$$E ::= [\,] \mid E \star M \mid n \star E \mid EN \mid \mathsf{seq}\ E\ M \mid \mathsf{ifz}\ E\ \mathsf{then}\ M\ \mathsf{else}\ N$$
$$\mid \mathsf{assign}\ M\ E \mid \mathsf{assign}\ E\ n \mid \mathsf{deref}\ E \mid \mathsf{cas}\ L\ E\ N \mid \mathsf{cas}\ L\ m\ E \mid \mathsf{cas}\ E\ m\ n$$

$$\langle E[m \star n], s\rangle \longrightarrow \langle E[n'], s\rangle, \text{ where } n' = m \star n \qquad \langle E[(\lambda x^{\mathtt{T}}.M)N], s\rangle \longrightarrow \langle E[M[N/x]], s\rangle$$
$$\langle E[\mathsf{ifz}_{\mathtt{T}}\ 0\ \mathsf{then}\ M\ \mathsf{else}\ N], s\rangle \longrightarrow \langle E[M], s\rangle \qquad \langle E[\mathsf{ifz}_{\mathtt{T}}\ n+1\ \mathsf{then}\ M\ \mathsf{else}\ N], s\rangle \longrightarrow \langle E[N], s\rangle$$
$$\langle E[\mathsf{fix}_{\mathtt{T}}\ M], s\rangle \longrightarrow \langle E[M(\mathsf{fix}_{\mathtt{T}}\ M)], s\rangle \qquad \langle E[\mathsf{seq}_{\mathtt{T}}\ \mathsf{skip}\ M], s\rangle \longrightarrow \langle E[M], s\rangle$$
$$\langle E[\mathsf{assign}\ v\ n, s\rangle] \longrightarrow \langle E[\mathsf{skip}], \langle s \mid v \mapsto n\rangle\rangle \qquad \langle E[\mathsf{deref}\ v], s\rangle \longrightarrow \langle E[s(v)], s\rangle$$
$$\langle E[\mathsf{cas}\ v\ m\ n], s\rangle \longrightarrow \langle E[n], \langle s \mid v \mapsto n\rangle\rangle, \text{ where } s(v) = m$$
$$\langle E[\mathsf{cas}\ v\ m\ n], s\rangle \longrightarrow \langle E[s(v)], s\rangle, \text{ where } s(v) \neq m$$
$$\langle E[\mathsf{assign}\ (\mathsf{mkvar}\ M\ N\ L)\ n], s\rangle \longrightarrow \langle E[Mn], s\rangle \qquad \langle E[\mathsf{deref}\ (\mathsf{mkvar}\ M\ N\ L)], s\rangle \longrightarrow \langle E[N], s\rangle$$
$$\langle E[\mathsf{cas}\ (\mathsf{mkvar}\ M\ N\ L)\ m\ n], s\rangle \longrightarrow \langle E[Lmn], s\rangle$$
$$\langle E[M\ \mathsf{or}\ N], s\rangle \longrightarrow \langle E[M], s\rangle \qquad \langle E[M\ \mathsf{or}\ N], s\rangle \longrightarrow \langle E[N], s\rangle$$
$$\langle E[\mathsf{newvar}\ x = n\ \mathsf{in}\ \mathsf{skip}], s\rangle \longrightarrow \langle E[\mathsf{skip}], s\rangle$$

$$\frac{\langle C, \langle s \mid x \mapsto n\rangle\rangle \longrightarrow \langle C', s'\rangle}{\langle E[\mathsf{newvar}\ x = n\ \mathsf{in}\ C], s\rangle \longrightarrow \langle E[\mathsf{newvar}\ x = s'(x)\ \mathsf{in}\ C'], \langle s' \mid x \mapsto s(x)\rangle\rangle}$$

$$\langle E[\mathsf{parallel}\ x_1 = k_1, \ldots, x_m = k_m\ \mathsf{in}\ \mathsf{skip} \| \mathsf{skip}\ \mathsf{finally}\ Q], s\rangle \longrightarrow \langle E[\mathsf{newvar}\ x_1 = k_1\ \mathsf{in}\ \cdots\ \mathsf{newvar}\ x_m = k_m\ \mathsf{in}\ Q], s\rangle$$

$$\frac{\langle P_j, \langle s \mid x_i \mapsto k_i\rangle\rangle \longrightarrow \langle P'_j, s'\rangle \qquad P'_{1-j} = P_{1-j} \qquad j = 0 \text{ or } 1}{\langle E[\mathsf{parallel}\ x_1 = k_1, \ldots, x_m = k_m\ \mathsf{in}\ P_0 \| P_1\ \mathsf{finally}\ Q], s\rangle \longrightarrow \langle E[\mathsf{parallel}\ x_1 = s'(x_1), \ldots, x_m = s'(x_m)\ \mathsf{in}\ P'_0 \| P'_1\ \mathsf{finally}\ Q], s\rangle}$$

Fig. 2 Evaluation context and 1-step reduction rules.

M *may-converges* at initial state s, if $\langle M, s\rangle \longrightarrow^* \langle V, s'\rangle$ holds for some canonical form V and state s'. Also, we say the evaluation of a program M at the initial state s *must-converges*, if there is a sufficiently large number m such that no 1-step reduction sequence starting from $\langle M, s\rangle$ is longer than m. In particular when M is a closed term, we write $M \Downarrow^{\mathrm{may}}$ to mean that M may-converges to some canonical value V and also write $M \Downarrow^{\mathrm{must}}$ to mean that M must-converges (for arbitrary initial state).

A *contextual preorder* on terms is defined by means of both may- and must-convergences. We define $M \lesssim_{\mathrm{may}} N$ iff $C[M] \Downarrow^{\mathrm{may}}$ implies $C[N] \Downarrow^{\mathrm{may}}$; $M \lesssim_{\mathrm{must}} N$ iff $C[M] \Downarrow^{\mathrm{must}}$ implies $C[N] \Downarrow^{\mathrm{must}}$; $M \lesssim_{\mathrm{m\&m}} N$ iff $M \lesssim_{\mathrm{may}} N$ and $M \lesssim_{\mathrm{must}} N$.

The language $\mathsf{EIA}_{\mathsf{par}}$ extends Idealized Algol with some non-sequential language constructs, erratic nondeterminism M_1 or M_2 (which nondeterministically chooses one out of the two, as introduced in EIA [7], [8]), and the parallelism.

The newvar construct introduces a mutable local variable and its scope. The local variables can be accessed by atomic operations: read, write, and *compare-and-set* (*CAS*). The value stored in a variable v is read by deref v and is updated by assign $v\ n$ with a new value n. A CAS operation cas v *old new* is a memory operation supported by most modern CPU architectures for mutual exclusion and it *atomically* executes like a program:

newvar t in (t :=!v; if !t = *old* then v := *new*; *new* else !t).

In words, it updates variable v with the value *new* only if v stores the presupposed value *old*; otherwise it leaves the variable v untouched. In both cases, it returns the value of v at the moment

of finishing the operation. Notice that the above successive sequence of memory operations are executed as a single atomic operation that is uninterruptible.

In addition to local variables, the language also provides bad-variables (mkvar), whose response to each variable access operation can be arbitrarily defined. Bad-variables are needed to establish the full abstraction result [3], [4], [8].

The parallelism in $\mathsf{EIA}_{\mathsf{par}}$ is expressed by a parallel execution command $\mathsf{parallel}\ x_1 = k_1, \ldots, x_m = k_m\ \mathsf{in}\ P_0 \| P_1\ \mathsf{finally}\ Q$, where two commands[*1] P_0 and P_1 run in parallel with interleaved access to the shared variables $x_1, \ldots, x_m$. The *wrap-up command* Q is executed immediately after both parallel processes have terminated. One should notice that, due to the typing constraint in Fig. 1, each parallel process P_i is prohibited to access identifiers in the context Γ and is allowed to refer to only those locally declared shared variables $x_1, \ldots, x_m$. In process calculi, the command would be roughly expressed by a term $(\nu\vec{x})((P_0 \mid P_1); Q)$, where $\vec{x}$ is the set of channels representing the declared set of shared variables, and no channels other than $\vec{x}$ is free in P_0 or P_1. Thus the channels $\vec{x}$ are shared by P_0, P_1, and Q but not by the surrounding context; The channels other than $\vec{x}$ are shared by Q and the surrounding context, but not by P_0 or P_1. (This peculiar form of parallel command is due to some technical reason, which we will discuss later in Section 4.)

The typing constraint above prohibits us from writing parametrized parallel programs, e.g., procedures. This does not rule out parametrized parallel programs all together, however. Given a parametrized program, we can still reason about it by instantiating every parameter with a particular instance.

Further, though the shared variables are not directly accessible from outside the local scope, their values can be made accessible via the wrap-up command Q, e.g.,

$$w : \mathtt{var} \vdash \mathsf{parallel}\ v = 0\ \mathsf{in}\ v :=\!!v + 1 \| v :=\!!v + 1\ \mathsf{finally}\ w :=\!!v$$

where the wrap-up command $w :=\!!v$ saves the value of the shared variable v in w for possible future uses in the surrounding context.

3. Games for Nondeterminism

This section provides the background matters on the game theoretic accounts needed in the following sections. Here we only give a digest of the whole details, mostly following Harmer's thesis [8], which is based on Hyland-Ong style dialogue games [2]. See Refs. [4], [13] for more general accounts on the game semantics.

In the rest of this paper, we use the following notations for finite strings (or sequences) over some alphabet. An empty string is denoted by ε. Given two strings s and t, we write st or $s \cdot t$ for the concatenation and $s \sqsubseteq t$ for the prefix ordering, meaning that $s \cdot u = t$ for some u. We also write $s \upharpoonright K$ for the *restriction* of s to a set K of alphabets, i.e., the substring of s consisting of all occurrences of alphabets in K. Given a set T of finite strings, T^{even} (resp., T^{odd}) denotes the subset of T containing solely the even (resp., odd) length strings.

[*1] It is easy to allow arbitrary number of processes running in parallel, but for simplicity we only consider execution of two parallel processes in this paper.

3.1 Arenas

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

- M_A is a set of *moves*.
- $\lambda_A : M_A \to \{\mathsf{O}, \mathsf{P}\} \times \{\mathsf{Q}, \mathsf{A}\}$, called *labeling* function, assigns each move $m \in M_A$ its attributes, either opponent(O) or player(P) and either question(Q) or answer(A). For brevity, let us write $\lambda_A^{\mathsf{OP}}(m)$ (resp., $\lambda_A^{\mathsf{QA}}(m)$) for the opponent/player (resp., question/answer) attribution of the move m.
- $\vdash_A$ is a binary relation on $M_A \times M_A$, called *enabling*, satisfying

(e1) $(n \vdash_A m \wedge n \neq m) \implies \lambda_A^{\mathsf{OP}}(n) \neq \lambda_A^{\mathsf{OP}}(m)$
(e2) $m \vdash_A m \implies (\lambda_A(m) = (\mathsf{O}, \mathsf{Q}) \wedge (n \neq m \implies n \not\vdash_A m))$
(e3) $(n \vdash_A m \wedge \lambda_A^{\mathsf{QA}}(m) = \mathsf{A}) \implies \lambda_A^{\mathsf{QA}}(n) = \mathsf{Q}$

When $n \vdash m$, where $n \neq m$, we say *n justifies m*. Justification relation strictly alternates opponent/player (e1) and every move that justify an answer move must be a question move (e3). A move m satisfying $m \vdash_A m$ is called an *initial move*. Every initial move m must be a question played by the opponent and is never justified by other moves (e2).

For notational convenience, given an arena A, we write $\overline{\lambda}_A$ for the labeling function whose opponent/player attribution is swapped, i.e., $\overline{\lambda}_A^{\mathsf{OP}}(m) = \mathsf{O}$ iff $\lambda_A^{\mathsf{OP}}(m) = \mathsf{P}$ for every move m. Also, we write $\mathsf{Q}(A)$ for the set of question moves in A and $\mathsf{A}_q(A)$ for the set of answer moves responding to the question move q.

A *justified string* in an arena A is a sequence of moves, in which every non-initial move m has a pointer to an earlier occurrence of a justifying move n (i.e., $n \vdash m$), written like $\cdots\ n \overleftarrow{\cdots}\ m\ \cdots$. Also, an initial move n is said to *hereditarily justify m*, if there is a chain of justification pointers starting from m and reaching n. We write $s \upharpoonright n$ for the subsequence of s consisting of all those occurrences of moves hereditarily justified by n.

In our game based model, every opponent move must be immediately followed by a player move, if any. We call a justified string s a *legal play* iff s is a string like $o_1 p_1 o_2 p_2 \cdots$ that strictly alternates opponent moves o_i's and player moves p_i's. The set of legal plays in arena A is denoted by L_A.

Here we give a few basic arenas.

- The arena $\mathbf{1} = \langle \emptyset, \emptyset, \emptyset \rangle$ is a trivial arena whose only legal play is an empty sequence.
- The arena $\mathbf{N}$ of *natural numbers* consists of $M_{\mathbf{N}} = \{\mathtt{q}\} \cup \{n \mid n = 0, 1, 2, \ldots\}$, $\lambda_{\mathbf{N}}(\mathtt{q}) = (\mathsf{O}, \mathsf{Q})$, $\lambda_{\mathbf{N}}(n) = (\mathsf{P}, \mathsf{A})$, and $\mathtt{q} \vdash_{\mathbf{N}} \mathtt{q}$ and $\mathtt{q} \vdash_{\mathbf{N}} n$ for all n.
- The arena $\mathbf{C}$ of *commands* consists of $M_{\mathbf{C}} = \{\mathtt{run}, \mathtt{done}\}$, $\lambda_{\mathbf{C}}(\mathtt{run}) = (\mathsf{O}, \mathsf{Q})$, $\lambda_{\mathbf{C}}(\mathtt{done}) = (\mathsf{P}, \mathsf{A})$, and $\mathtt{run} \vdash_{\mathbf{C}} \mathtt{run}$ and $\mathtt{run} \vdash_{\mathbf{C}} \mathtt{done}$.

In the arena $\mathbf{N}$ of natural numbers, a legal play, say, $\mathtt{q} \overleftarrow{\cdot}\ 8$ models an interaction where the opponent first asks for a natural number and the player answers 8 as a response. In the arena $\mathbf{C}$, a legal play $\mathtt{run} \overleftarrow{\cdot}\ \mathtt{done}$ models a similar interaction, but the player answers done to the opponent's question run, just signaling the termination of the execution of command.

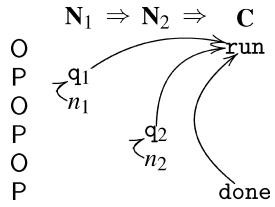
Existing arenas can be combined to form new arenas. Below we write $[\lambda_A, \lambda_B]$ for the copairing function over the disjoint sum $M_A + M_B$ of moves from two given arenas A and B, that is, $[\lambda_A, \lambda_B](m) = \lambda_A(m)$ (resp., $[\lambda_A, \lambda_B](m) = \lambda_B(m)$) if m is an

injected move of A (resp., B).

- A *product arena*, written $A \times B$, is a triple $\langle M_{A\times B}, \lambda_{A\times B}, \vdash_{A\times B}\rangle$, where $M_{A\times B} = M_A + M_B$, $\lambda_{A\times B} = [\lambda_A, \lambda_B]$, and $n \vdash_{A\times B} m$ iff $n \vdash_A m \vee n \vdash_B m$.
  Note that empty arena $\mathbf{1}$ is the unit of the product construction.
- An *arrow arena*, written $A \Rightarrow B$, is a triple $\langle M_{A\Rightarrow B}, \lambda_{A\Rightarrow B}, \vdash_{A\Rightarrow B}\rangle$, where $M_{A\Rightarrow B} = M_A + M_B$, $\lambda_{A\Rightarrow B} = [\overline{\lambda}_A, \lambda_B]$, and $n \vdash_{A\Rightarrow B} m$ iff $n \vdash_B m \vee (n \neq m \wedge n \vdash_A m) \vee (n \vdash_B n \wedge m \vdash_A m)$.

For example, the following legal play in arena $\mathbf{N}_1 \Rightarrow \mathbf{N}_2 \Rightarrow \mathbf{C}$

models an execution of a command that terminates after evaluating its first natural number argument and then the second one. (Subscripts are attached to distinguish different copies of the same arena and the corresponding moves.) Notice that the opponent/player moves in the *contravariant* arena $\mathbf{N}$ are switched. This is better illustrated in the following vertical representation.



In what follows, justification pointers are omitted, as long as they can be inferred with no ambiguity.

3.2 Strategies

A *strategy* σ in arena A is a pair (T_σ, D_σ) of subsets of legal plays. T_σ is a subset of L_A^{even}, called *traces*, satisfying (t1) $\varepsilon \in T_\sigma$ and (t2) $sab \in T_\sigma \implies s \in T_\sigma$. That is, T_σ is an even-length prefix closed subset of L_A^{even}, by which it is specified how player can respond to each opponent's move. Below we write $dom(\sigma)$ for $\{sa \in L_A^{\mathrm{odd}} \mid \exists b.sab \in T_\sigma\}$ and *contingency closure* $cc(\sigma)$ for $T_\sigma \cup dom(\sigma)$. For every $sa \in L_A^{\mathrm{odd}}$, we define $rng_\sigma(sa) = \{b \mid sab \in T_\sigma\}$.

The remaining component D_σ, called *divergences*, is a subset of L_A^{odd} satisfying the following properties.

(d1) $(s \in T_\sigma \wedge sa \in L_A \wedge sa \notin dom(\sigma)) \implies \exists d \in D_\sigma.d \sqsubseteq sa$,
(d2) $sa \in D_\sigma \implies s \in T_\sigma$, and
(d3) For every $sa \in dom(\sigma)$ such that $rng_\sigma(sa)$ has an infinite cardinality, $\exists d \in D_\sigma.d \sqsubseteq sa$.

When a subsequence $sa \in D_\sigma$ has seen in a game play, it means that the corresponding program execution must have diverged. (Every divergence must be reached by a trace (d2).) We only consider finitely branching strategies, i.e., infinitely many player's choices indicate divergence (d3). Also, if an odd-length legal play sa does not allow player to respond, i.e., $sa \notin dom(\sigma)$, it must be a divergence, called *uninteresting* divergence (d1).

A play $s \in T_\sigma$ is called *complete* if every question move in s has been answered by an answer move. The set of complete traces in σ is denoted by $comp(\sigma)$.

In practice, we will consider a further confined class of strategies satisfying (*Player*) *visibility* and *bracketing* conditions. See Refs. [7], [8] for the precise definition for these conditions.

3.3 Composing Strategies

For any strategies $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$, we write $\sigma;\tau$ for the composition of the two strategies. The composition $\sigma;\tau$ gives rise to a strategy in the arena $A \Rightarrow C$, where the interactions on the arena B have been hidden.

A *legal interaction* of arenas A, B, and C is a finite string u of the moves from the three arenas, where u has justification pointers as specified in $A \Rightarrow B$ and $B \Rightarrow C$. We denote the set of legal interactions by $int(A, B, C)$.

Given $u \in int(A, B, C)$, we write $u \upharpoonright B, C$ for a legal play in the arena $B \Rightarrow C$ obtained by removing all A moves and those relevant pointers that justify A moves. Similarly, we write $u \upharpoonright A, B$ for the corresponding legal play in the arena $A \Rightarrow B$. We also write $u \upharpoonright A, C$ for a legal play in the arena $A \Rightarrow C$ obtained by removing all B moves and relevant pointers but with one exception: whenever $u = \cdots\ c \overset{\cdots}{\leftarrow} b \overset{\cdots}{\leftarrow} a\ \cdots$ for some $a \in M_A$, $b \in M_B$, $c \in M_C$, we replace the two justification pointers by a single pointer from a to c.

When two strategies $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$ are composed, the resulting traces are those obtained by hiding all the B-moves occurring in the interaction of the two strategies. That is, the traces are defined by:

$$T_{\sigma;\tau} = \{u \upharpoonright A, C \mid u \in T_\sigma \parallel T_\tau\},$$

where $T_\sigma \parallel T_\tau = \{u \in int(A, B, C) \mid u \upharpoonright A, B \in T_\sigma \wedge u \upharpoonright B, C \in T_\tau\}$.

For the divergence, there are three different possible patterns. A divergence is triggered either by a move of σ, by a move of τ, or by a livelock caused by *infinite chattering*, i.e., an infinite sequence of interaction between σ and τ that contains infinitely many moves of B but only finite observable moves of A and C.

Suppose u^∞ is a string of an infinite length. Let us write $u' \sqsubseteq^{\mathrm{fin}} u^\infty$ to mean that u' is a finite prefix of u^∞. An *infinite interaction* u^∞ is an infinite string whose every finite prefix is contained in $int(A, B, C)$. The set of infinite interactions is denoted by $int^\infty(A, B, C)$.

The set of divergences is defined as a union of three subsets with the restriction to the moves of A and C:

$$D_{\sigma;\tau} = \{u \upharpoonright A, C \mid u \in T_\sigma \not\downarrow D_\tau \cup D_\sigma \not\downarrow T_\tau \cup T_\sigma \not\downarrow T_\tau\},$$

where the first and second ones are *finitely generated* divergences:

$$T_\sigma \not\downarrow D_\tau = \{u \in int(A, B, C) \mid u \upharpoonright A, B \in T_\sigma \wedge u \upharpoonright B, C \in D_\tau\},$$
$$D_\sigma \not\downarrow T_\tau = \{u \in int(A, B, C) \mid u \upharpoonright A, B \in D_\sigma \wedge u \upharpoonright B, C \in T_\tau\},$$

and the last one is *infinitely generated* divergences:

$$T_\sigma \not\downarrow T_\tau = \{u^\infty \in int^\infty(A, B, C) \mid u^\infty \upharpoonright A, C \in L_{A\Rightarrow C} \wedge \forall u' \sqsubseteq^{\mathrm{fin}} u^\infty.(u' \upharpoonright A, B \in cc(\sigma) \wedge u' \upharpoonright B, C \in cc(\tau))\}.$$

3.4 The Category of Games

For every non-empty legal play $sm \in L_A$, let us define *current thread* $\lceil sm \rceil$ by $sm \upharpoonright n$, where n is the initial move that hereditarily justifies m. In words, n is the move that initiates the (single) parallel thread and $\lceil sm \rceil$ comprises of the moves that are relevant to the thread execution.

We are concerned with a collection of interleaved execution paths of multiple parallel threads, where executions of threads are independent of each other. Such a collection can be identified by a class of *single-threaded* strategy. A strategy $\sigma = (T_\sigma, D_\sigma)$ is called single-threaded if it satisfies the following conditions.

- For every $sab \in T_\sigma$, b is justified by a move in $\lceil sa \rceil$,
- For every sab, $t \in T_\sigma$, if $ta \in L_A^{\mathrm{odd}}$ and $\lceil sa \rceil = \lceil ta \rceil$, then $tab \in T_\sigma$ and tab has justification pointer from b to an earlier occurrence of move such that $\lceil sab \rceil = \lceil tab \rceil$,
- $\forall s \in T_\sigma.(sa \in D_\sigma \implies \exists d \sqsubseteq sa.\lceil d \rceil \in D_\sigma)$, and
- $\forall s \in T_\sigma.(\lceil sa \rceil \in D_\sigma \implies \exists d \sqsubseteq sa.d \in D_\sigma)$.

Given strategies σ and τ, the lower ordering $\leq^\flat$ for traces, the upper ordering $\leq^\sharp$ for divergences, and the convex ordering $\leq^\natural$ for both are defined by:

- $\sigma \leq^\flat \tau$ iff $T_\sigma \subseteq T_\tau$.
- $\sigma \leq^\sharp \tau$ iff $\forall e \in D_\tau.\exists d \in D_\sigma.d \sqsubseteq e$ and $\forall sab.(sab \in T_\tau \wedge sab \notin T_\sigma \implies \exists d \in D_\sigma.d \sqsubseteq sab)$.
- $\sigma \leq^\natural \tau$ iff $\sigma \leq^\flat \tau \wedge \sigma \leq^\sharp \tau$.

The ordering $\sigma \leq^\natural \tau$ should be read: τ is more likely to terminate than σ. Let us write $\sigma =^\natural \tau$ for the derived equality, i.e., $\sigma =^\natural \tau$ iff $\sigma \leq^\natural \tau$ and $\tau \leq^\natural \sigma$.

We have a symmetric monoidal category $\mathcal{G}$, whose objects are arenas and arrows are strategies in arrow arenas (i.e., an arrow $A \to B$ is a strategy in the arena $A \Rightarrow B$) satisfying the visibility and bracketing conditions. The identity arrow $\mathtt{id}_A : A \to A$ is the *copycat strategy*, in which every opponent move is immediately copied by a player move. Formally, $\mathtt{id}_A$ is a strategy that has empty divergences and has traces:

$$T_{\mathtt{id}_A} = \{s \in L^{\mathrm{even}}_{A \Rightarrow A} \mid \forall s' \in L^{\mathrm{even}}_{A_1 \Rightarrow A_2}.s' \sqsubseteq s \text{ implies } s' \upharpoonright A_1 = s' \upharpoonright A_2\},$$

where $s' \upharpoonright A_i$ $(i = 1, 2)$ denotes a restriction of s' to a subsequence of moves in the arena A_i.

We give the game theoretic model in a cartesian closed category $\mathcal{C}$, which is a lluf subcategory of $\mathcal{G}$ obtained by restricting arrows to those single-threaded strategies modulo $=^\natural$. The category $\mathcal{C}$ is also CPO-enriched w.r.t. the ordering $\leq^\natural$. In what follows, we denote the product of two arrows $f : A \to B$ and $g : A \to C$ by $\langle f, g \rangle : A \to B \times C$. Also, given $f : B \times A \to C$, we write $\Lambda_A(f) : B \to A \Rightarrow C$ for the mapping corresponding to the currying natural isomorphism.

Finally to note, it is known that single-threaded strategies have a bijective correspondence with so-called well-opened strategies up to $=^\natural$. A strategy σ is called *well-opened* iff its traces are well-opened, i.e., every $s \in T_\sigma$ has exactly one initial move. The bijective correspondence is established by a pair of mappings $WO(-)$ and $ST(-)$ such that $\sigma =^\natural ST(WO(\sigma))$ and $WO(ST(\upsilon)) =^\natural \upsilon$ for every single-threaded strategy σ and well-opened strategy υ [8].

By this, we can uniquely identify a single-threaded strategy σ by a pair (T, D) of well-opened traces T and interesting divergences D. The corresponding single-threaded strategy can be obtained as $ST((T, D'))$, where D' is the superset of D containing all uninteresting divergences induced by (d1).

4. The Game Semantics and Full Abstraction

We present a game semantics for the language $\mathsf{EIA}_{\mathsf{par}}$ and show the full abstraction result. To avoid verbosity, we will concentrate mostly on the modeling of parallelism and related matters without giving the details of the proof. Interested readers are deferred to the first author's thesis [14].

4.1 Modeling Memory Access

Let us first define the arena **Var** of stores, corresponding to the type `var`.

- $M_{\mathbf{Var}} = \{\mathtt{rd}\} \cup \{\mathtt{wr}_n \mid n \geq 0\} \cup \{\mathtt{cas}_{m,n} \mid m, n \geq 0\} \cup \{n \mid n \geq 0\} \cup \{\mathtt{ok}\}$,
- $\lambda_{\mathbf{Var}}(\mathtt{rd}) = \lambda_{\mathbf{Var}}(\mathtt{wr}_n) = \lambda_{\mathbf{Var}}(\mathtt{cas}_{m,n}) = (\mathsf{O}, \mathsf{Q})$ and $\lambda_{\mathbf{Var}}(n) = \lambda_{\mathbf{Var}}(\mathtt{ok}) = (\mathsf{P}, \mathsf{A})$, and
- For every m, n, k, $\mathtt{rd} \vdash_{\mathbf{Var}} \mathtt{rd}$, $\mathtt{rd} \vdash_{\mathbf{Var}} n$, $\mathtt{wr}_n \vdash_{\mathbf{Var}} \mathtt{wr}_n$, $\mathtt{wr}_n \vdash_{\mathbf{Var}} \mathtt{ok}$, $\mathtt{cas}_{m,n} \vdash_{\mathbf{Var}} \mathtt{cas}_{m,n}$, and $\mathtt{cas}_{m,n} \vdash_{\mathbf{Var}} k$.

Each memory operation can be modeled as interaction with the arena **Var**. The CAS operation is modeled by

$$[\![\mathtt{cas}\ L\ M\ N]\!] = \langle [\![L]\!], [\![M]\!], [\![N]\!] \rangle; \mathtt{cas},$$

where $\mathtt{cas} : \mathbf{Var} \times \mathbf{N}_1 \times \mathbf{N}_2 \Rightarrow \mathbf{N}_3$ is the corresponding strategy that has no divergences and traces $\{\mathtt{q}_3 \cdot \mathtt{q}_1 \cdot m_1 \cdot \mathtt{q}_2 \cdot n_2 \cdot \mathtt{cas}_{m,n} \cdot l \cdot l_3 \mid l, m, n \geq 0\}$. Each play first asks the second argument M for a number m, then asks the third argument N for a number n, and finally returns the number l returned by interacting with the store by issuing the CAS operation $\mathtt{cas}_{m,n}$ to the store. The models of remaining memory operations are standard. The read operation is modeled by $[\![\mathtt{deref}\ M]\!] = [\![M]\!]; \mathtt{deref}$, where $\mathtt{deref} : \mathbf{Var} \Rightarrow \mathbf{N}$ is the strategy with traces $\{\mathtt{q} \cdot \mathtt{rd} \cdot n \cdot n \mid n \geq 0\}$. The write operation is modeled by $[\![\mathtt{assign}\ M\ N]\!] = \langle [\![M]\!], [\![N]\!] \rangle; \mathtt{assign}$, where $\mathtt{assign} : \mathbf{Var} \times \mathbf{N} \Rightarrow \mathbf{C}$ is the strategy with traces $\{\mathtt{run} \cdot \mathtt{q} \cdot n \cdot \mathtt{wr}_n \cdot \mathtt{ok} \cdot \mathtt{done} \mid n \geq 0\}$.

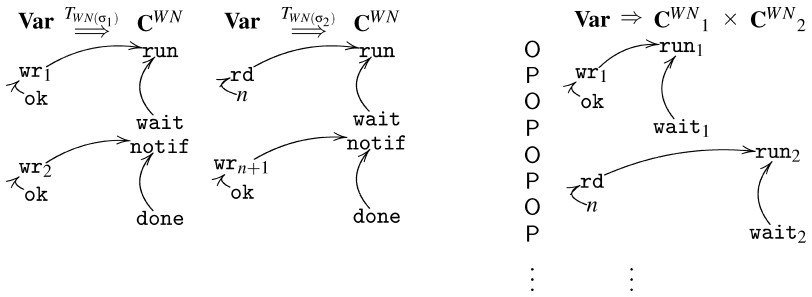
Notice that memory access operations have been modeled as if they were interacting with a volatile memory: A read on a store can return an arbitrary value rather than the one stored by the last write or CAS. In the following, we say that a legal play in **Var** is a *successful write by n* if it has the form either $\mathtt{wr}_n \cdot \mathtt{ok}$ or $\mathtt{cas}_{m,n} \cdot n$ $(n, m \geq 0)$.

We define $\mathtt{cell}_i$ as the strategy in **Var** for a single memory cell that initially stores the value i and constrains the moves subject to the causality implied by the order of memory operations. Formally, $\mathtt{cell}_i$ is a strategy such that $D_{\mathtt{cell}_i} = \emptyset$ and every trace $s \cdot a \cdot b \in T_{\mathtt{cell}_i}$ satisfies either of the following conditions.

(i) $a = \mathtt{wr}_n$ and $b = \mathtt{ok}$.
(ii) $a = \mathtt{rd}$ and $b = i$, if s contains no successful write.
(iii) $a = \mathtt{rd}$ and $b = n$, if the last successful write in s is by n.
(iv) $a = \mathtt{cas}_{m,n}$ and $b = n$, if either s contains no successful write and $m = i$ or the last successful write in s is by m.
(v) $a = \mathtt{cas}_{m,n}$ and $b = i$, if s contains no successful write and $m \neq i$.
(vi) $a = \mathtt{cas}_{m,n}$ and $b = k$, if the last successful write in s is by k and $m \neq k$.

Newvar. The local variable scoping can be modeled in the standard way [3], by constraining game interaction in the arena **Var** subject to the cell strategy.

Given $\Gamma = x_1 : \mathtt{T}_1, \ldots, x_k : \mathtt{T}_k$, let us write $[\![\Gamma]\!]$ for the object $[\![\mathtt{T}_1]\!] \times \cdots \times [\![\mathtt{T}_k]\!]$, where each $[\![\mathtt{T}_i]\!]$ is the arena corresponding to $\mathtt{T}_i$, and $!_{[\![\Gamma]\!]}$ for the unique arrow $[\![\Gamma]\!] \to \mathbf{1}$. Writing $\mathtt{Cell}_{\Gamma,n}$ for the arrow $!_{[\![\Gamma]\!]}; \mathtt{cell}_n : [\![\Gamma]\!] \to \mathbf{Var}$, we can model the newvar



(a) Wait-notify lifting (b) Interleaved play

Fig. 3 Interleaving by wait-notify lifting.

construct as follows.

$$[\![\Gamma \vdash \mathsf{newvar}\ x = n\ \mathsf{in}\ M]\!] = ST\left(\langle \mathtt{id}_{[\![\Gamma]\!]}, \mathtt{Cell}_{\Gamma,n}\rangle; WO\left(\Lambda^{-1}_{\mathbf{Var}}([\![\lambda x^{\mathrm{var}}.M]\!])\right)\right)$$

4.2 Modeling Parallelism

Let us consider the game semantics for the parallel command. For simplicity, we only consider parallel commands with a single shared variable. The general cases can be easily derived.

Two processes running in parallel can interleave only between their atomic variable accesses: Once a process issued a memory operation, another process has to wait for the operation to finish. For example, let us try to model the interleaved execution of a parallel program:

$$\mathsf{parallel}\ v = 0\ \mathsf{in}\ (v := 1; v := 2) \| (v := !v + 1).$$

The parallel processes have traces: $T_{\sigma_1} = \{\mathtt{run}\cdot\mathtt{wr}_1\cdot\mathtt{ok}\cdot\mathtt{wr}_2\cdot\mathtt{ok}\cdot\mathtt{done}\}$ and $T_{\sigma_2} = \{\mathtt{run}\cdot\mathtt{rd}\cdot n\cdot\mathtt{wr}_{n+1}\cdot\mathtt{ok}\cdot\mathtt{done} \mid n \geq 0\}$. One might expect to have the following play as the result of interleaving:

	Var	⇒	$\mathbf{C}_1$	×	$\mathbf{C}_2$
O			$\mathtt{run}_1$		
P	$\mathtt{wr}_1$				
O	ok				
P					$\underline{\mathtt{run}_2}$
O	$\underline{\mathtt{rd}}$				
P	$\underline{n}$				
⋮		⋮			

where the underlines are put to distinguish moves from different parallel threads. However, the above interleaving is not legal, since $\underline{\mathtt{run}_2}$ is an opponent move but it occurs in a player position. This violates the parity condition that opponent/player alternate.

To solve this problem, we insert auxiliary `wait`/`notif` moves between each successive atomic memory operations in each game play, as shown in **Fig. 3** (a), so that they can be interleaved without breaking the parity condition.

We call this construction of strategies *wait-notify lifting*. The lifted strategies allow us to interleave atomic memory operations without violating parity condition, as shown in Fig. 3 (b). Parallel processes can switch contexts wherever a `wait` move is immediately followed by a `notif` move: Each `wait` move signals that the current process yields its shared variable access to the other running process and each `notif` move indicates that the current process is notified to be able to resume its shared variable access.

Suppose that $\sigma : \mathbf{Var} \Rightarrow \mathbf{C}$ is a strategy that models one of parallel processes that share a single variable. The wait-notify lifting of σ, written $WN(\sigma)$, is a strategy in $\mathbf{Var} \Rightarrow \mathbf{C}^{WN}$, where $\mathbf{C}^{WN}$ is a new arena that extends $\mathbf{C}$ with moves `wait` and `notif`.

- $M_{\mathbf{C}^{WN}} = \{\mathtt{run}, \mathtt{done}, \mathtt{wait}, \mathtt{notif}\}$,
- $\lambda_{\mathbf{C}^{WN}}(m) = (\mathsf{O}, \mathsf{Q})$ iff $m \in \{\mathtt{run}, \mathtt{notif}\}$, $\lambda_{\mathbf{C}^{WN}}(m) = (\mathsf{P}, \mathsf{A})$ iff $m \in \{\mathtt{done}, \mathtt{wait}\}$, and
- $\mathtt{run} \vdash_{\mathbf{C}^{WN}} \mathtt{run}$, $\mathtt{notif} \vdash_{\mathbf{C}^{WN}} \mathtt{notif}$, $\mathtt{run} \vdash_{\mathbf{C}^{WN}} \mathtt{done}$, $\mathtt{run} \vdash_{\mathbf{C}^{WN}} \mathtt{wait}$, $\mathtt{notif} \vdash_{\mathbf{C}^{WN}} \mathtt{wait}$, and $\mathtt{notif} \vdash_{\mathbf{C}^{WN}} \mathtt{done}$.

The trace part $T_{WN(\sigma)}$ of the lifted strategy is defined as the least even-length prefix closed set that satisfies the following conditions.

- $\mathtt{run}\cdot\mathsf{q}_1\cdot\mathsf{a}_1\cdots\mathsf{q}_i\cdot\mathsf{a}_i\cdot\mathsf{q}_{i+1}\cdot\mathsf{a}_{i+1}\cdot\cdots\cdot\mathsf{a}_{k-1}\cdot\mathsf{q}_k \in T_\sigma$ implies $\mathtt{run}\cdot\mathsf{q}_1\cdot\mathsf{a}_1\cdot\mathtt{wait}\cdot\mathtt{notif}\cdots\mathsf{q}_i\cdot\mathsf{a}_i\cdot\mathtt{wait}\cdot\mathtt{notif}\cdot\mathsf{q}_{i+1}\cdot\mathsf{a}_{i+1}\cdot\cdots\cdot\mathtt{wait}\cdot\mathtt{notif}\cdot\mathsf{a}_{k-1}\cdot\mathsf{q}_k \in T_{WN(\sigma)}$, where each $\mathsf{q}_i\cdot\mathsf{a}_i$ is a legal play in $\mathbf{Var}$ and the last q_k is either $\mathtt{rd}, \mathtt{wr}_n, \mathtt{cas}_{m,n}$ or `done`.
- Every answer move (i.e., `wait` or `done`) has a justification pointer to the last preceding occurrence of question move (i.e., `run` or `notif`).

The divergences $D_{WN(\sigma)}$ consists of those obtained by interspersing $\mathtt{wait}\cdot\mathtt{notif}$ in the plays from D_σ (similarly as above) and also uninteresting divergences deduced by (d1).

To interleave strategies $\tau_1, \tau_2 : \mathbf{Var} \Rightarrow \mathbf{C}$, representing two processes running in parallel, wherever context switching is allowed, we define the "interleaved" strategy $IL(\tau_1, \tau_2) : \mathbf{Var} \Rightarrow \mathbf{C}$ as:

$$IL(\tau_1, \tau_2) = \Delta_{\mathbf{Var}}; ST((WN(WO(\tau_1)) \times WN(WO(\tau_2)); \mathtt{pmerge}),$$

where $\Delta_A = \langle \mathtt{id}_A, \mathtt{id}_A\rangle$ is the usual diagonal strategy $A \Rightarrow A \times A$ and `pmerge` is a strategy for generating all possible interleaving. The strategy $\mathtt{pmerge} : \mathbf{C}^{WN}{}_1 \times \mathbf{C}^{WN}{}_2 \Rightarrow \mathbf{C}$ is generated by the pair $(T, \emptyset)$ of well-opened traces and (empty) divergences such that

$$T = \{s_1 \bowtie s_2 \mid s_i \in \mathtt{run}_i \cdot (\mathtt{wait}_i \cdot \mathtt{notif}_i)^* \cdot \mathtt{done}_i\ (i = 1, 2)\},$$

where $s_1 \bowtie s_2$ denotes *bigram shuffle*, the set of possible interleaving of two strings at their even positions (i.e., $w \bowtie \varepsilon = \varepsilon \bowtie w = \{w\}$, $(a\cdot b\cdot w) \bowtie (a'\cdot b'\cdot w') = \{a\cdot b\cdot s \mid s \in w \bowtie (a'\cdot b'\cdot w')\} \cup \{a'\cdot b'\cdot s \mid s \in (a\cdot b\cdot w) \bowtie w'\}$, and $a \bowtie b = \emptyset$).

Parallel. The parallel command is then modeled by the interleaved execution of parallel processes, followed by the execution of the wrap-up command Q.

$$[\![\Gamma \vdash \mathsf{parallel}\ x = k\ \mathsf{in}\ P_1 \| P_2\ \mathsf{finally}\ Q : \mathtt{com}]\!] =$$
$$ST\Big(\langle \mathtt{id}_{[\![\Gamma]\!]}, \mathtt{Cell}_{\Gamma,k}\rangle; WO(\langle !_{[\![\Gamma]\!]} \times \mathtt{id}_{\mathbf{Var}}; IL([\![P_1]\!], [\![P_2]\!]),$$
$$\Lambda^{-1}_{\mathbf{Var}}([\![\lambda x^{\mathtt{var}}.Q]\!])\rangle; \mathtt{seqC}\Big).$$

4.3 Full Abstraction

The full abstraction result for our parallel language $\mathsf{EIA}_{\mathsf{par}}$ is obtained basically by a similar discussion on single-threaded strategies in Ref. [8], but we have to be careful about the compositionality of parallel commands.

A bare parallel command $P_1 \| P_2$, which simply executes in parallel with sharing all the local variables in the context, would not compose with other language constructs to give a correct game model. That means, the following standard lemma would *not* hold.

Substitution lemma If $\Gamma, x : \mathtt{T}_2 \vdash M : \mathtt{T}_1$ and $\Gamma \vdash N : \mathtt{T}_2$ are well-typed terms then so is $\Gamma \vdash M[N/x] : \mathtt{T}_1$ and $[\![M[N/x]]\!] = \langle \mathtt{id}_{[\![\Gamma]\!]}, [\![N]\!]\rangle; [\![M]\!]$.

For example, let us consider a bare parallel command $v : \mathtt{var}, c : \mathtt{com} \vdash c \| c : \mathtt{com}$. Since the abstracted command c carries no concrete information how it will access the shared variable v when instantiated, there is no chance of interleaving variable accesses and so the program will be modeled as equal to $v : \mathtt{var}, c : \mathtt{com} \vdash c; c : \mathtt{com}$. However, substituting $v :=\!v + 1$ to c in both programs, we obtain two instances of programs $v : \mathtt{var}, c : \mathtt{com} \vdash v :=\!v+1 \| v :=\!v + 1 : \mathtt{com}$ and $v : \mathtt{var}, c : \mathtt{com} \vdash v :=\!v + 1; v :=\!v + 1 : \mathtt{com}$, which should apparently be modeled as different programs.

The problem is that, when parallel processes contain references to those other than the shared variables, instantiations to those references may add extra accesses to the shared variables, but these extra accesses cannot be properly interleaved because no concrete information about instantiation is available for the identifiers yet to be instantiated. So we delimit the scope of each parallel command with a designated list of shared variables, disallowing references to other identifiers. With this syntactic constraints, simple induction suffices for proving the substitution lemma.

4.3.1 Soundness

The soundness theorem can be proved in a similar way of Refs. [7], [8], by showing that the consistency and adequacy properties.

Proposition 4.1 (consistency) Suppose that M is a closed term of type $\mathtt{com}$. If $M \Downarrow^{\mathrm{may}}$ then $\mathtt{run} \cdot \mathtt{done} \in T_{[\![M]\!]}$ and if $M \Downarrow^{\mathrm{must}}$ then $\mathtt{run} \notin D_{[\![M]\!]}$.

Proposition 4.2 (adequacy) Suppose that M is a closed term of type $\mathtt{com}$. If $\mathtt{run} \cdot \mathtt{done} \in T_{[\![M]\!]}$ then $M \Downarrow^{\mathrm{may}}$ and if $\mathtt{run} \notin D_{[\![M]\!]}$ then $M \Downarrow^{\mathrm{must}}$.

The consistency can be proved by induction on the length of reduction sequences. The adequacy follows from the fact that every program satisfies the *computability* predicate [8], but we have to alter the original definition of the predicate to incorporate the interleaved execution of parallel processes.

Theorem 4.3 (soundness) If M and N are closed terms of $\mathsf{EIA}_{\mathsf{par}}$ such that $[\![M]\!] \preceq^{\natural} [\![N]\!]$ then $M \lesssim_{\mathrm{m\&m}} N$.

4.3.2 Definability and Full Abstraction

For the converse of the soundness theorem, we have to show that a certain class of strategies are definable in $\mathsf{EIA}_{\mathsf{par}}$ terms.

Proposition 4.4 Suppose that $\sigma : [\![\Gamma]\!] \Rightarrow [\![T]\!]$ is a compact strategy satisfying visibility and bracketing conditions. Then, σ is definable in $\mathsf{EIA}_{\mathsf{par}}$ without parallel constructs.

This result can be shown in a similar way to Ref. [8]. The only difference is that we need to take care of CAS operations, but this has only a minor impact on the entire proof.

Here we notice that parallel constructs are not needed to show the definability result. This implies that the erratic nondeterminism already has the enough power for expressing the shared variable parallelism in $\mathsf{EIA}_{\mathsf{par}}$ [*2].

The full abstraction result follows from the definability result, as usual, for a certain quotient model, called *intrinsic collapse*. Given two strategies $f, g : A \to B$, let us define a preorder $f \preccurlyeq g$ holds iff 'f'; $t \preceq^{\natural}$ 'g'; t for every *testing strategy* $t : A \Rightarrow B \to \mathbf{C}$, where '$f$' : $\mathbf{1} \to A \Rightarrow B$ denotes the obvious currying of $f : \mathbf{1} \times A \to B$. We write $f \simeq g$ for the derived equivalence relation, i.e, $f \simeq g$ iff $f \preccurlyeq g$ and $g \preccurlyeq f$. Let us write $\mathcal{E}[\![M]\!]$ for the intrinsic quotient of M, i.e., the equivalence class that contains $[\![M]\!]$ in $C/\simeq$.

The intrinsic quotient model is intended to distinguish programs by testing their termination behavior under arbitrary context of type $\mathtt{com}$. There are three possible outcomes of testing: definite termination (i.e., $\mathtt{run} \cdot \mathtt{done} \in T_\tau$ and $D_\tau = \emptyset$), both of possible termination and possible divergence (i.e., $\mathtt{run} \cdot \mathtt{done} \in T_\tau$ and $\mathtt{run} \in D_\tau$), and definite divergence (i.e., $T_\tau = \emptyset$ and $\mathtt{run} \in D_\tau$).

Theorem 4.5 (full abstraction) For every closed terms M and N of $\mathsf{EIA}_{\mathsf{par}}$, $\mathcal{E}[\![M]\!] \preccurlyeq \mathcal{E}[\![N]\!]$ iff $M \lesssim_{\mathrm{m\&m}} N$.

5. Checking Contextual Equivalence of Parallel Programs

In this section, we will argue how the equivalence of parallel programs can be determined in the game model. Unfortunately the equivalence of programs is known to be an undecidable problem for the full IA language, but it becomes decidable when restricted to certain classes of sublanguages [15]. Following Ref. [9], we will consider a sublanguage that restricts the full language $\mathsf{EIA}_{\mathsf{par}}$ to the second-order (i.e., only first-order types occur in terms and typing contexts), finitary (i.e., only finitely many varieties of moves are observed) [*3], and iterative (i.e., recursion is only available in iterative form $\mathsf{whilenz}\ M\ \mathsf{do}\ C$, which repeatedly executes command C until M evaluates to 0) fragments. Then the program equivalence can be argued as the equality over regular languages, as done by Ghica and McCusker for IA [9].

In what follows, we will use the following notations for regular languages. In addition to the usual regular expressions (null language ϕ, null string ε, single alphabet a, concatenation $R \cdot R'$, union $R + R'$, and Kleene closure R^*), we may use the following several extensions: $\sum_{i \in I} R_i$ denotes a finite union over a finite index set I; $R \cap R'$ denotes intersection, the set of common strings

[*2] We do not insist that every form of parallelism is expressible by nondeterminism. Recall that the parallel construct in $\mathsf{EIA}_{\mathsf{par}}$ is subject to the syntactic constraint discussed above.

[*3] In this section, we assume a finite set of natural numbers, say, the set of numbers representable in a machine word.

contained in both languages; $R_1 \bowtie R_2$ denotes the bigram shuffle (Section 4.2) of two regular languages; Hiding operator $R \downharpoonleft B$ removes any occurrences of alphabets contained in B; Broadening operator $\widetilde{R}$ intersperses any alphabets other than those occurring in the language of R at arbitrary positions of strings.

In this section, we identify the strategy τ of a program P by a pair (T_P, D_P) of extended regular expressions, where the even-length prefixes of T_P specifies the well-opened traces of τ and D_P specifies the interesting divergences. Furthermore, following Ref. [9], we will systematically annotate game moves with superscripted labels as $(-)^{\langle \ell \rangle}$. For example, the trace part of the game interpretation $[\![c : \mathtt{var}_0 \to \mathtt{var}_1 \to \mathtt{com}, v_0 : \mathtt{var}, v_1 : \mathtt{var} \vdash c\, v_0\, v_1 : \mathtt{com}]\!]$ is expressed by the regular expression $\mathscr{V}$:

$$\mathscr{V} = \mathtt{run}^{\langle c \rangle} \cdot \Big(\sum_{j \in \{0,1\}} \sum_{q \in \mathbf{Q}(\mathbf{Var})} \sum_{a \in \mathsf{A}_q(\mathbf{Var})} q^{\langle jc \rangle} \cdot q^{\langle v_j \rangle} \cdot a^{\langle v_j \rangle} \cdot a^{\langle jc \rangle} \Big)^* \cdot \mathtt{done}^{\langle c \rangle},$$

where $m^{\langle c \rangle}$ stands for the **C** moves of c and $m^{\langle jc \rangle}$ for the **Var** moves of c accessing the $(j+1)$-th argument. Similarly, $m^{\langle v_j \rangle}$ stands for the moves for accessing identifier v_j.

Recall the parallel program we considered in Introduction. The implementation I in the $\mathsf{EIA}_{\mathsf{par}}$ syntax is given as below.

$$I \equiv \mathsf{parallel}\ b = 0, f_0 = 0, f_1 = 0, v_0 = 0, v_1 = 0 \\ \mathsf{in}\ P_0 \| P_1\ \mathsf{finally}\ c\, v_0\, v_1,$$

where $P_i \equiv f_i := 1; (\mathsf{whilenz}\ !f_{1-i}\ \mathsf{do\ skip}); v_i :=\!!b; b := 1 - !b; f_i := 0$, for each $i = 0, 1$. Here the wrap-up command $c\, v_0\, v_1$ intends an execution of an arbitrary command that processes on the shared variables v_0 and v_1.

Let us show that I is an incorrect implementation by discriminating it from the following *specification* program S, which makes use of nondeterminism to express the original intention of the program.

$$S \equiv \mathsf{newvar}\ v_0 = 0, v_1 = 0 \\ \mathsf{in}\ (((v_0 := 0; v_1 := 1)\ \mathsf{or}\ (v_1 := 0; v_0 := 1));\ c\, v_0\, v_1)$$

The trace part of the while loop in each parallel thread P_i is characterized by the regular language $\mathtt{run} \cdot L_i \cdot \mathsf{done}$ where

$$L_i = \Big(\sum_{n \neq 0} (\mathtt{rd}^{\langle f_{1-i} \rangle} \cdot n^{\langle f_{1-i} \rangle}) \Big)^* \cdot \mathtt{rd}^{\langle f_{1-i} \rangle} \cdot 0^{\langle f_{1-i} \rangle},$$

which models an iterated read access to the variable f_{1-i} until 0 is returned as the result. No divergence is witnessed at this moment, as the arena **Var** itself poses no causality between individual memory operations and thus there is no evidence of divergence.

Each thread P_i has the traces $\mathtt{run} \cdot T_i \cdot \mathsf{done}$ where

$$T_i = \mathtt{wr}_1^{\langle f_i \rangle} \cdot \mathtt{ok}^{\langle f_i \rangle} \cdot L_i \\ \cdot \sum_{n,m} (\mathtt{rd}^{\langle b \rangle} \cdot n^{\langle b \rangle} \cdot \mathtt{wr}_n^{\langle v_i \rangle} \cdot \mathtt{ok}^{\langle v_i \rangle} \cdot \mathtt{rd}^{\langle b \rangle} \cdot m^{\langle b \rangle} \cdot \mathtt{wr}_{1-m}^{\langle b \rangle} \cdot \mathtt{ok}^{\langle b \rangle}) \\ \cdot \mathtt{wr}_0^{\langle f_i \rangle} \cdot \mathtt{ok}^{\langle v_i \rangle},$$

and thus the traces of the implementation I is given by:

$$T_I = \Big((\mathtt{run} \cdot (T_0 \bowtie T_1) \cdot \mathscr{V} \cdot \mathsf{done}) \\ \cap \widetilde{\gamma_0^b} \cap \widetilde{\gamma_0^{f_0}} \cap \widetilde{\gamma_0^{f_1}} \cap \widetilde{\gamma_0^{v_0}} \cap \widetilde{\gamma_0^{v_1}} \Big) \downharpoonleft U',$$

where $U' = \{m^{\langle v \rangle} \mid m \in M_{\mathbf{Var}}, v \in \{b, f_0, f_1, v_0, v_1\}\}$ and γ_0^v denotes the regular language of the strategy $\mathtt{cell}_0$, whose every move is annotated with $\langle v \rangle$.

By a similar discussion, we can express the traces of the specification program S by the regular expression:

$$T_S = \Big(\Big(\mathtt{run} \cdot \Big(\sum_{i \in \{0,1\}} \mathtt{wr}_0^{\langle v_i \rangle} \cdot \mathtt{ok}^{\langle v_i \rangle} \cdot \mathtt{wr}_1^{\langle v_{1-i} \rangle} \cdot \mathtt{ok}^{\langle v_{1-i} \rangle} \Big) \\ \cdot \mathscr{V} \cdot \mathsf{done} \Big) \cap \widetilde{\gamma_0^{v_0}} \cap \widetilde{\gamma_0^{v_1}} \Big) \downharpoonleft U,$$

where $U = \{m^{\langle v \rangle} \mid m \in M_{\mathbf{Var}}, v \in \{v_0, v_1\}\}$.

It is easy to see that T_S and T_I denote the same language, since every subsequence of variable access in each language that precedes $\mathscr{V}$ leaves the variable v_0 and v_1 assigned with values 0 and 1 (in this or reversed order) when the sequence is restricted w.r.t. the cell strategies. This implies that I is may-equivalent to S.

However, they are not may&must-equivalent. The language $T_0 \bowtie T_1$ contains arbitrarily long sequences:

$$\mathtt{wr}_1^{\langle f_0 \rangle} \cdot \mathtt{ok}^{\langle f_0 \rangle} \cdot \mathtt{wr}_1^{\langle f_1 \rangle} \cdot \mathtt{ok}^{\langle f_1 \rangle} \cdot \mathtt{rd}^{\langle f_{i_1} \rangle} \cdot 1^{\langle f_{i_1} \rangle} \cdot \mathtt{rd}^{\langle f_{i_2} \rangle} \cdot 1^{\langle f_{i_2} \rangle} \cdots$$

where i_k's are either 0 or 1. This means that a divergence is generated by infinite chattering when the cell strategy is composed with the interleaved strategy of parallel processes. Thus I and S are discriminated by divergences, $D_I = \mathtt{run}$ and $D_S = \phi$.

In general, discriminating divergences is not sufficient for refuting may&must-equivalence, since strategies should be distinguished upto intrinsic collapse (Theorem 4.5). Murawski [15] has shown that the equivalence modulo intrinsic collapse can be effectively judged in terms of *winning region* of reachability games. The winning region (for player) in a strategy τ is the set of odd-length legal plays from which the player can, against any proceeding counter moves by opponent, make choices of moves to construct a complete play that has no diverging prefixes. Formally, it is an infinite union of $\bigcup_{i=0}^{\infty} W_i$, where $W_0 = \{s \cdot o \mid \exists p.s \cdot o \cdot p \in \mathit{comp}(\tau) \text{ and } \forall d \in D_\tau.d \not\sqsubseteq s \cdot o\}$ and $W_{i+1} = W_i \cup \{s \cdot o \mid \exists p. \forall o'.s \cdot o \cdot p \cdot o' \in W_i\}$ $(i \geq 0)$. Then, $M_1 \lesssim_{\mathrm{must}} M_2$ iff, for every $s \in (T_2 \cup D_2) \setminus (T_1 \cup D_1)$ where $(T_i, D_i) = [\![M_i]\!]$ $(i = 1, 2)$, there exists an odd-length prefix s' of s that is *not* in the winning region of $[\![M_1]\!]$.

Applying this to our example, we can see that $S \not\lesssim_{\mathrm{must}} I$, because we have $\mathtt{run} \in (T_{[\![I]\!]} \cup D_{[\![I]\!]}) \setminus (T_{[\![S]\!]} \cup D_{[\![S]\!]})$ but $\mathtt{run}$, which is the sole odd-length prefix of itself, is in the winning region of $[\![S]\!]$.

In order to give a correct parallel implementation, which is may&must-equivalent to S, we may use CAS operations for mutual exclusion. Let us define a program I' that is obtained from I by replacing each of its subcommand P_i with P_i', where

$$P_i' \equiv \mathsf{newvar}\ t = 0\ \mathsf{in}\ (t :=\!!b; v_i := \mathsf{cas}\ b\ !t\ (1 - !t)).$$

By a similar discussion as above, we can show that the regular language model of this implementation I' has no divergences and has traces:

$$T_{I'} = \Big(\Big((\mathtt{run} \cdot (T_0' \bowtie T_1') \cdot \mathsf{done}) \\ \cap \widetilde{\gamma_0^b} \cap \widetilde{\gamma_0^{f_0}} \cap \widetilde{\gamma_0^{f_1}} \cap \widetilde{\gamma_0^{v_0}} \cap \widetilde{\gamma_0^{v_1}} \Big) \Big) \downharpoonleft U',$$

where $T'_i = \sum_{n,m}\left(\mathtt{rd}^{\langle b\rangle}\cdot n^{\langle b\rangle}\cdot \mathtt{cas}^{\langle b\rangle}_{n,1-n}\cdot m^{\langle b\rangle}\cdot \mathtt{wr}^{\langle v_i\rangle}_{m}\cdot \mathtt{ok}^{\langle v_i\rangle}\right)$ for each $i = 0, 1$.

This language of traces is equal to that of S's, since, for every possible interleaving expressed by $T'_0 \bowtie T'_1$, the process P_i that first executes the CAS operation witnesses the moves $\mathtt{cas}^{\langle b\rangle}_{0,1}\cdot 1^{\langle b\rangle}\cdot \mathtt{wr}^{\langle v_i\rangle}_{1}\cdot \mathtt{ok}^{\langle v_i\rangle}$ while the other witnesses $\mathtt{cas}^{\langle b\rangle}_{1,0}\cdot 0^{\langle b\rangle}\cdot \mathtt{wr}^{\langle v_{1-i}\rangle}_{0}\cdot \mathtt{ok}^{\langle v_{1-i}\rangle}$. Since both P'_0 and P'_1 have no divergences and their traces are all finite, $D_{I'} = \phi$. Therefore we conclude that $S \simeq_{\mathrm{m\&m}} I'$.

6. Conclusion

We proposed a fully abstract game semantics for an Algol-like shared variable parallel programming language. In contrast to existing proposals, our game semantics observes divergence as well as termination and thus is more appropriate for detecting possible unpreferable behavior of parallel programs. Our game semantics is built on the work for erratic nondeterminism [7], [8], and the full abstraction result indicates that the parallel command adds no extra power than erratic nondeterminism.

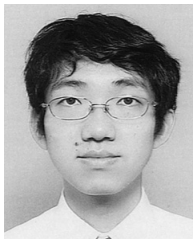
We believe that the procedure for checking the may&must-equivalence of parallel programs can be automated, as done in Ref. [9]. We hope to report some progress on this topic elsewhere in the future.

A more challenging research topic would be a development of parallel semantics for the so called relaxed (weak) memory models, in which memory operations may be reordered for the sake of higher performance [16]. Although the strong memory model is assumed thoughout the paper, the game semantical approach might contribute to deeper understanding of the relaxed memory models, by appropriately interpreting the relaxed constraint by nondeterminism.

Acknowledgments We would like to thank the reviewer, whose comments and suggestions were valuable for improvement of the paper.

References

[1] Herlihy, M. and Shavit, N.: *The Art of Multiprocessor Programming*, Morgan Kaufmann (2008).
[2] Hyland, J.M.E. and Ong, C.-H.L.: On Full Abstraction for PCF: I, II, and III, *Information and Computation*, Vol.163, No.2, pp.285–408 (2000).
[3] Abramsky, S. and McCusker, G.: Linearity, Sharing and State: A Fully Abstract Game Semantics for Idealized Algol with Active Expressions, *Algol-like Languages*, O'Hearn, P.W. and Tennent, R.D. (Eds.), Progress in Theoretical Computer Science, Vol.2, pp.297–329, Birkhäuser (1997).
[4] Abramsky, S. and McCusker, G.: Game Semantics, *Computational Logic: Proc. 1997 Marktoberdorf Summer School*, Schwichtenberg, H. and Berger, U. (Eds.), pp.1–56, Springer-Verlag (1999).
[5] Ghica, D.R. and Murawski, A.S.: Angelic Semantics of Fine-Grained Concurrency, *Annals of Pure and Applied Logic*, Vol.151, No.2-3, pp.89–114 (2008).
[6] Ghica, D.R., Murawski, A.S. and Ong, C.-H.L.: Syntactic Control of Concurrency, *Theor. Comput. Sci.*, Vol.350, No.2-3, pp.234–251 (2006).
[7] Harmer, R. and McCusker, G.: A Fully Abstract Game Semantics for Finite Nondeterminism, *Proc. 14th Annual IEEE Symposium on Logic in Computer Science*, pp.422–430, IEEE Computer Society (1999).
[8] Harmer, R.: Games and Full Abstraction for Nondeterministic Languages, PhD Thesis, University of London (1999).
[9] Ghica, D.R. and McCusker, G.: The Regular-Language Semantics of Second-Order Idealized ALGOL, *Theor. Comput. Sci.*, Vol.309, No.1–3, pp.469–502 (2003).
[10] Hoare, C.A.R.: *Communicating Sequential Processes*, Prentice Hall (1985).
[11] Milner, R.: *Communicating and Mobile Systems: The π-calculus*, Cambridge University Press (1999).
[12] Brookes, S.: Full Abstraction for a Shared Variable Parallel Language, *Proc. 8th Annual IEEE Symposium on Logic in Computer Science*, pp.98–109, IEEE Computer Society (1993).
[13] Abramsky, S.: Algorithmic Game Semantics: A Tutorial Introduction, *Proof and System Reliability*, Schichtenberg, H. and Steinbruggen, R. (Eds.), pp.21–47, Kluwer Academic (2001).
[14] Watanabe, K.: Full Abstraction for an Algol-Like Language with Shared Variable Parallelism, Master's thesis, Dept. Math., Faculty of Sci., Kyoto University (2012). available from ⟨http://www.math.kyoto-u.ac.jp/~susumu/papers/watanabe2012msc/⟩.
[15] Murawski, A.: Reachability Games and Game Semantics: Comparing Nondeterministic Programs, *Proc. 23rd Annual IEEE Symposium on Logic in Computer Science*, pp.353–363, IEEE Computer Society Press (2008).
[16] Sorin, D.J., Hill, M.D. and Wood, D.A.: *A Primer on Memory Consistency and Cache Coherence*, Morgan & Claypool (2011).



Keisuke Watanabe graduated from Division of Mathematics and Mathematical Analysis, Kyoto University and received his master's degree from Kyoto University in 2012. From 2012 spring, he works as a math teacher at Tezukayama Junior and Senior High School.

Susumu Nishimura received his bachelor's degree from Kyoto University in 1992. He received his master's and doctor's degrees from Faculty of Science, Kyoto University in 1994 and 1996, respectively. Since then, he worked as a research associate in RIMS, Kyoto University. Since 2003, he was appointed as an associate professor of the same university, at Graduate School of Science. His research interest is program transformation and, in general, theory of programming languages. He is a member of Japan Society for Software Science and Technology.