Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT
Theoretical Computer Science 309 (2003) 469–502

Theoretical Computer Science

www.elsevier.com/locate/tcs

The regular-language semantics of second-order idealized ALGOL

Dan R. Ghica[a,*], Guy McCusker[b]

[a] *Department of Computing and Information Science, Queen's University, Kingston, Ont., Canada K7L 3N6*

[b] *School of Cognitive and Computing Sciences, University of Sussex at Brighton, Falmer, Brighton, BN1 9QH, UK*

Received 13 November 2001; accepted 1 May 2003
Communicated by D. Sannella

Abstract

We explain how recent developments in game semantics can be applied to reasoning about equivalence of terms in a non-trivial fragment of Idealized ALGOL (IA) by expressing sets of complete plays as regular languages. Being derived directly from the fully abstract game semantics for IA, our model inherits its good theoretical properties; in fact, for second-order IA taken as a stand-alone language the regular language model is fully abstract. The method is algorithmic and formal, which makes it suitable for automation. We show how reasoning is carried out using a meta-language of extended regular expressions, a language for which equivalence is decidable.

Keywords: Game semantics; ALGOL-like languages; Regular languages

1. Introduction

Reynolds's Idealized ALGOL (IA) is a compact language which combines the fundamental features of imperative languages with a full higher-order procedure mechanism. This combination makes the language very expressive. For example, simple forms of classes and objects may be encoded in IA [28]. For these reasons, IA has attracted a great deal of attention from theoreticians; some 20 papers spanning almost 20 years of research were recently collected in book form [22].

* Corresponding author.
E-mail address: dan.ghica@comlab.ox.ac.uk (D.R. Ghica).

0304-3975/$ - see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/S0304-3975(03)00315-3

A common theme in the literature on semantics of IA, beginning with [16], is the use of putative program equivalences to test suitability of semantic models. These example equivalences are intended to capture intuitively valid principles such as the privacy of local variables, irreversibility of state changes and representation independence. A good model should support these intuitions.

Over the years, a variety of models have been proposed, each of which went some way towards formalizing programming intuition: functor categories gave an account of variable allocation and deallocation [24], relational parametricity was employed to capture representation-independence properties [21], and linear logic to explain irreversibility [20]. Recently, many of these ideas have been successfully incorporated in an operationally based account of IA by Pitts [26].

A frustrating situation was created with the development of a fully abstract game semantics for IA [1]. The full abstraction result means that the model validates all (and only) correct equivalences between terms, but unfortunately the model as originally presented is complicated, and calculating and reasoning within the model is difficult.

In this article, we show how by restricting attention to the second-order subset of IA, the games model can be simplified dramatically: terms now denote regular languages, and a relatively straightforward notation can be used to describe and calculate with the simplified semantics. The fragment of IA which we consider contains almost all the example equivalences from the literature, and we are able to validate them in a largely calculational, algebraic style. We also obtain a decidability result for equivalence of programs in this fragment.

The approach of game semantics, and therefore of this paper, has little in common with the traditional semantics of IA. Intuitively, it comes closest to Reddy's "object semantics" [27] and Brookes's trace semantics for shared-variable concurrent ALGOL [3]. Although the language has assignment, a semantic notion of store is not used; although the language has procedures, a semantic notion of function is not used. Instead, we are primarily concerned with behaviour, with all the possible actions that can be associated with every language entity. Meanings of phrases are then constructed combinatorially according to the semantic rules of the language.

We believe our new presentation of game semantics is elementary enough to be considered a potential "popular semantics" [32]; it should at least provide a point of entry to game semantics for those who have previously found the subject opaque. Moreover, the property of full abstraction together with the fact that reasoning can be carried out in a decidable formal language suggest that our approach constitutes a good foundation on which an automatic program checker for IA and related languages can be constructed (see also [5,7]). The idea of using game semantics to support automated program analysis has already been independently explored in a more general framework by Hankin and Malacaria [9,10]. They used such models to derive static analysis algorithms which can be described without reference to games.

In the following section, we will describe the fragment of IA we are addressing. After that, we provide a very gentle and informal introduction to the games model of IA. Section 4 presents the regular language model for the language fragment. In Section 5, we illustrate our model with several important putative equivalences. In the subsequent section, we prove that this model of let-free IA is a correct representa-

tion of the games model, then in Section 7, we address the issue of adding function definitions.

2. The programming language fragment

The principles of the programming language IA were laid down by John Reynolds in an influential paper [29]. IA is a language that combines imperative features with a procedure mechanism based on a typed call-by-name lambda calculus; local variables obey a stack discipline, having a lifetime dictated by syntactic scope; expressions, including procedures returning a value, cannot have side effects, i.e. they cannot assign to non-local variables. We conform to these principles, except for the last one. This flavour of IA is known as IA with *active expressions* and has been analysed extensively [34,1,20]. We consider only the recursion-free second-order fragment of this language, the fragment which has been used to give virtually all the significant equivalences mentioned in the literature. In addition, we will only deal with finite data sets.

The data types τ of the language (i.e. types of data assignable to variables) are a finite subset of the integers and the booleans. The phrase types of the language are those of commands, variables and expressions, plus first-order function types.

$$\tau ::= \mathbf{int} \mid \mathbf{bool},$$
$$\sigma ::= \mathbf{comm} \mid \mathbf{var}\,\tau \mid \mathbf{exp}\,\tau,$$
$$\theta ::= \sigma \mid \sigma \to \theta.$$

Terms are introduced using type judgements of the form:

$$\Gamma \vdash M : \sigma, \quad \Gamma = \{x_1 : \theta_1, \ldots, x_k : \theta_k\},$$

where the x_j are drawn from a countable set of *identifiers*.

We will be concerned with proving equivalences of the form

$$\Gamma \vdash M \equiv_\sigma M'.$$

The terms of the language and their typing rules are presented in Fig. 1.

The data types of the language, i.e. the types of values assignable to variables, are bounded integers (**int**) and booleans (**bool**). The phrase types, i.e. the types of terms, are commands (**comm**), boolean and integer variables (**varint**, **varbool**) and expressions (**expint**, **expbool**), as well as first-order functions. The usual operators of arithmetic and logic are employed $(- * -)$.

The imperative constructs are the common ones: assignment $(:=)$, command sequencing $(;)$, iteration (**while**) and branching (**if**). Other common branching (**case**) and iterative constructs (**for**, **do-until**) are not included because they can be easily expressed in terms of the existing ones. They do not contribute semantically, being only what is called *syntactic sugar*. Branching is imposed uniformly on types, so we have branching for expressions (similar to the ?-:- operator in C), variable and function-typed terms. The behaviour of variables in imperative languages is dual, depending on whether they occur on the left-hand side (*l-values*) or right-hand side (*r-values*) of assignment

$$\frac{}{\Gamma \vdash \mathbf{skip} : \mathbf{comm}} \qquad \frac{}{\Gamma \vdash \mathbf{diverge}_\sigma : \sigma}$$

$$\frac{}{\Gamma \vdash b : \mathbf{expbool}}\ b \in \{\mathbf{true}, \mathbf{false}\} \qquad \frac{}{\Gamma \vdash n : \mathbf{expint}}\ n \in \mathscr{Z}$$

$$\frac{}{\Gamma, \iota : \theta \vdash \iota : \theta} \qquad \frac{\Gamma \vdash V : \mathbf{var}\,\tau}{\Gamma \vdash !V : \mathbf{exp}\,\tau}$$

$$\frac{\Gamma \vdash E_1 : \mathbf{exp}\,\tau \quad \Gamma \vdash E_2 : \mathbf{exp}\,\tau}{\Gamma \vdash E_1 * E_2 : \mathbf{exp}\,\tau'} \qquad \frac{\Gamma, \iota : \mathbf{var}\,\tau \vdash C : \mathbf{comm}}{\Gamma \vdash \mathbf{new}\,\tau\,\iota\,\mathbf{in}\,C : \mathbf{comm}}$$

$$\frac{\Gamma \vdash V : \mathbf{var}\,\tau \quad \Gamma \vdash E : \mathbf{exp}\,\tau}{\Gamma \vdash V := E : \mathbf{comm}} \qquad \frac{\Gamma \vdash C : \mathbf{comm} \quad \Gamma \vdash M : \sigma}{\Gamma \vdash C; M : \sigma}$$

$$\frac{\Gamma \vdash B : \mathbf{bool} \quad \Gamma \vdash M_1 : \sigma \quad \Gamma \vdash M_2 : \sigma}{\Gamma \vdash \mathbf{if}\ B\ M_1\ \mathbf{else}\ M_2 : \sigma} \qquad \frac{\Gamma \vdash B : \mathbf{expbool} \quad \Gamma \vdash C : \mathbf{comm}}{\Gamma \vdash \mathbf{while}\ B\ \mathbf{do}\ C : \mathbf{comm}}$$

$$\frac{\Gamma, \iota : \sigma \vdash P : \theta}{\Gamma \vdash \lambda\iota{:}\sigma.P : \sigma \to \theta} \qquad \frac{\Gamma \vdash P : \theta \quad \Gamma, \iota : \theta \vdash P' : \theta'}{\Gamma \vdash \mathbf{let}\ \iota\ \mathbf{be}\ P\ \mathbf{in}\ P'\ \mathbf{in} : \theta'}$$

$$\frac{\Gamma \vdash F : \sigma \to \theta \quad \Gamma \vdash M : \sigma}{\Gamma \vdash FM : \theta}$$

Fig. 1. Terms and typing rules of IA.

statements. The proper behaviour is usually automatically resolved by compilers using type-coercion rules, from variable types to expression types, when a variable is used on the right-hand side. For clarity of presentation we will not introduce such coercion rules, but we will use instead an explicit de-referencing operator (!) in the language. The main difference between the IA variant presented here and Reynolds's is that commands can be sequenced not only with commands but also with expressions or variables. The result is what is called an *active expression* (or variable). The informal semantics of an active expression is that it calculates a value while possibly writing to non-local variables. This is a common feature of most imperative languages. One special command of the language is **diverge**. It causes a program to enter an unresponsive state similar to that caused by an infinite loop. The command that performs no operation, similar to the empty command in C or PASCAL, is **skip**. For function declarations we use first-order lambda abstraction and a *let* constructor.

We call this self-contained programming language *second-order IA*.

2.1. Operational semantics of second-order IA

Our language has a standard operational semantics, given in terms of *stores*, functions from locations of type τ to values of type τ. The operational semantics is then an inductively defined relation of the form $s, M \Downarrow_\theta s', M'$, where M and M' are terms of type θ and s, s' are stores. We omit the definition here (see [1,26] for the details).

The notion of equivalence between terms is defined in the standard way. Given two terms $\Gamma \vdash M,\ M' : \theta$ we say that they are *observationally equivalent*, written $\Gamma \vdash M \equiv_\theta M'$

if and only if for any context $C[-]$ such that $\vdash C[M], C[M'] : \mathbf{comm}$, for all states s there exists state s' such that $s, C[M] \Downarrow_\theta s', \mathbf{skip}$ iff there exists s'' such that $s, C[M'] \Downarrow_\theta s'', \mathbf{skip}$.

We should remark that in the above definition, it makes no difference whether we consider contexts drawn from the second-order subset of IA on which we are focusing, or allow contexts to range over the full IA language. This is a consequence of Pitts's *Operational Extensionality* Theorem for IA [26], and can be demonstrated as follows.

Proposition 1 (Context restriction). *For all terms of second-order IA such that* $\Gamma \vdash P_1 \not\cong_\theta P_2$, *there is a context* $C[-]$ *such that* $\vdash C[P_1] \not\cong_{\mathbf{comm}} C[P_2]$ *with* $\vdash C[P_i] : \mathbf{comm}$ *closed terms of second-order IA.*

Proof (Outline). We denote extensional equivalence of IA terms [7, Definition 2.4] by $\cong$.

From the Operational Extensionality Theorem, $\Gamma \vdash P_1 \not\cong_\theta P_2$ iff $\Gamma \vdash P_1 \not\cong_\theta P_2$, which, using extensional equivalence of open terms is the case if and only if

$$W \vdash P_1[x_1, x_2, \ldots, x_n / P'_1, P'_2, \ldots, P'_n] \not\cong_\theta P_2[x_1, x_2, \ldots, x_n / P'_1, P'_2, \ldots, P'_n]$$

for some set of global variables $W = \{v_1, \ldots, v_k\}$ and terms P'_i of IA with global variables in W.

From the operational semantics of **let**, this means that discriminating contexts can be chosen to have the form:

$$\mathbf{new}\ v_1\ \mathbf{in} \cdots \mathbf{new}\ v_k\ \mathbf{in}\ \mathbf{let}\ x_1\ \mathbf{be}\ P'_1\ \mathbf{in} \cdots \mathbf{let}\ x_n\ \mathbf{be}\ P'_n\ \mathbf{in}\ [-].$$

Observation. Pitts's Operational Extensionality Theorem applies to IA without side effects in expressions, but in [13, Section 2.2] the first author shows that active IA also has this property. □

3. An informal introduction to the game semantics of IA

In game semantics, a computation is represented as an *interaction* between two protagonists: *Player* (P) represents the program, and *Opponent* (O) represents the environment or context in which the program runs. For example, for a program of the form $f : \mathbf{expint} \to \mathbf{comm} \vdash M : \mathbf{comm}$, *Player* will represent the program M; *Opponent* represents the context, in this case the non-local procedure f. This procedure, if called by M, may in turn call an argument, in which case O will ask P to provide this information.

The interaction between O and P consists of a sequence of moves, alternating between players. In the game for the type **comm**, for example, there is an initial move *run* to initiate a command, and a single response *done* to signal termination. Thus a simple interaction corresponding to the command **skip** is

O: *run* (start executing)
P: *done* (immediately terminate).

O: *run*	(start executing)
P: $run^{\langle f\rangle}$	(execute f)
O: $q^{\langle 1f\rangle}$	(what is the first argument to f?)
P: $0^{\langle 1f\rangle}$	(the argument is 0)
O: $done^{\langle f\rangle}$	(f terminates)
P: *done*	(whole command terminates).

Fig. 2. Typical play for term f : **expint** $\to$ **comm** $\vdash f(0)$: **comm**.

In more interesting games, such as the one used to interpret programs like f : **expint** $\to$ **comm** $\vdash f(0)$: **comm**, there are more moves. Corresponding to the result type **comm**, there are the moves *run* and *done*. The program needs to run the procedure f, so there are also moves $run^{\langle f\rangle}$ and $done^{\langle f\rangle}$ to represent that; here the $run^{\langle f\rangle}$ move is a move for P, and $done^{\langle f\rangle}$ is a move for O. Finally, the procedure f may need to evaluate its argument. For this purpose, O has a move $q^{\langle 1f\rangle}$, meaning "what is the value of the first argument to f?", to which P may respond with an integer n, tagged as $n^{\langle 1f\rangle}$ for the sake of identification.

In Fig. 2, we show a sample interaction in the interpretation of the above term. In this interaction, at the third move, O was not compelled to ask for the argument to f: if O represented a non-strict procedure, the move $done^{\langle f\rangle}$ would be played immediately. Similarly, at the fifth move, O could repeat the question $q^{\langle 1f\rangle}$ to represent a procedure which calls its argument more than once.

3.1. Strategies

Using the above ideas, each possible execution of a program is represented as a sequence of moves in the appropriate game. A *program* can therefore be represented as a *strategy* for P, i.e., a predetermined way of responding to the moves O makes. A strategy can also choose to make no response in a particular situation, representing divergence, so, for example, there are two strategies for the game corresponding to **comm**: the strategy for **skip** responds to *run* with *done*, and the strategy for **diverge** fails to respond to *run* at all.

Strategies are usually represented as *sets of sequences of moves*, so that a strategy is identified with the collection of possible traces that can arise if P plays according to that strategy. The fact that O can repeat questions, as we remarked above, means that these sets are very often infinite, even for simple programs. The strategy for the program $f(0)$, for example, is capable of supplying the argument 0 to f as often as O asks for it.

A property of strategies necessary for defining function application is *compositionality*. For example, consider a typical play of the addition operator, (Fig. 3) written in tabular format to indicate to what type occurrence every move belongs, together with the strategy for the constant pair $(3,5)$.

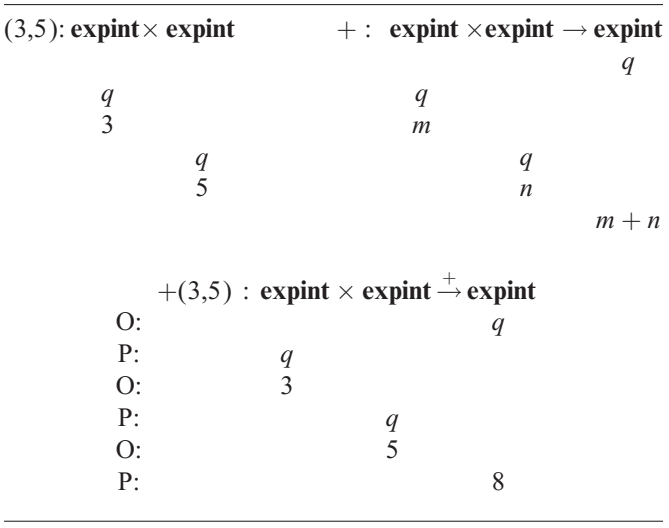


Fig. 3. Typical plays for the addition operator.

The addition operation $3 + 5$ is the application $+(3,5)$, and it is interpreted by composing the two strategies. The composition of strategies is defined by two stages: parallel composition followed by hiding. The parallel composition is the matching of the moves in the type of the arguments, **expint** × **expint**.

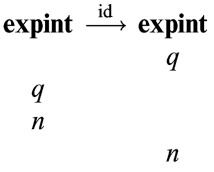
Hiding is the elimination of all the moves from the type of the arguments, resulting in

$3 + 5 = 8$: **expint**

O : q

P : 8

A certain special class of strategies plays an important role in modelling identity, abstraction, free identifiers and universal quantifiers. They are called *copy-cat* strategies, and are simply ways of copying information back and forth between the games. The typical play for the copy-cat strategy on **expint** is, for example,



3.2. Interpretation of variables

The type **var** τ is represented as a game in the following way. For each element v of τ there is an initial move *write*(v), representing an assignment. There is one possible response to this move, *ok*, which signals successful completion of the assignment.

O: *run*	
P: $read^{\langle v\rangle}$	(get the value from v)
O: $3^{\langle v\rangle}$	(O supplies the value 3)
P: $write(4)^{\langle v\rangle}$	(write 4 into v)
O: $ok^{\langle v\rangle}$	(the assignment is complete)
P: *done*	(the whole command is complete)

Fig. 4. Typical interaction for v: **varint** $\vdash v := !v + 1$.

For dereferencing, there is an initial move *read*, to which P may respond with any element of τ.

In Fig. 4, we show an interaction in the strategy for v: **varint** $\vdash v := !v + 1$.

In general, in these interactions, O is *not* constrained to exhibit the expected causal dependency between reads and writes. For example, in the game for terms of the form c: **comm**, v: **varint** $\vdash M$: **comm**, we may find interactions such as

$$run \cdot write(4)^{\langle v\rangle} \cdot ok^{\langle v\rangle} \cdot read^{\langle v\rangle} \cdot 3^{\langle v\rangle} \cdot run^{\langle c\rangle} \cdot done^{\langle c\rangle} \cdot read^{\langle v\rangle} \cdot 7^{\langle v\rangle} \cdots .$$

This play has two notable features:

- O has not played a good variable (in the sense of [30, Section 3.3.4]) in v, the value *read* immediately after a *write* being different than the written value. This can happen, for example, if the variable v stands for "bad variable" phrases.
- Two consecutive *read*s yielded different values, although no explicit *write*s to v intervened. This can happen if command c stands for a phrase that *interferes* with v, such as $v := 7$.

There is one situation in which neither of the above can happen: when the variable v is made local. This has two effects. The local interaction with v is guaranteed to exhibit good variable behaviour, and non-local entities cannot interfere with it. We shall call a variable which is both good and not interfered with (other than explicitly) a *stable variable*.

Also, the interaction with v is not an observable part of the program's behaviour. Therefore, the games interpretation of **new** τ v **in** M is given by taking the set of sequences interpreting M, considering only those in which O plays a good variable in v, then deleting all the moves pertaining to v, to hide it from the outside context.

3.3. Full abstraction

In [1], it was shown that games give rise to a fully abstract model of IA, in the following sense. Say that an interaction is *complete* if and only if it begins with an initial move and ends with a move which answers that initial move. Thus, for example, $run \cdot run^{\langle\iota\rangle}$ is not complete but $run \cdot run^{\langle\iota\rangle} \cdot done^{\langle\iota\rangle} \cdot done$ is. Then we have the following theorem:

Theorem 2 (Full abstraction for IA). *For any* $\Gamma \vdash P, Q : \theta$, *programs* P *and* Q *are contextually equivalent in IA* ($P \equiv Q$) *if and only if the sets of complete plays in the strategies interpreting* P *and* Q *are equal.*

In the above account, a very simple notion of game has been used. In fact, game models require a great deal more machinery, including the notions of *justification pointer* and *determinism*, in order for full abstraction to be achieved. The key observation which makes the present paper possible is that, for the interpretation of IA up to second-order types, this extra machinery is redundant. For a detailed account of the games model used here see [1,2,15].

4. Regular-language semantics

If we restrict IA to its recursion-free finitary second-order fragment, then much of the games apparatus becomes unnecessary. The justification pointers for all sets of complete plays of strategies are uniquely determined by the plays themselves, so need not be explicitly represented. Moreover, these sets are regular and can be described by a meta-language of extended regular expressions.

4.1. Lexical, alphabet and language operations

Several *lexical* operations are first needed. They involve tagging a symbol or changing the tagging of a symbol, resulting in a new symbol:

Definition 3 (Lexical operations).

Tag. Given two symbols $\alpha_0, \alpha \in \mathscr{A}$, $\alpha_0^{\langle\alpha\rangle}$ is a new symbol obtained by *tagging* the former with the latter.

We define the alphabet $\mathscr{A}^{\langle\alpha\rangle} = \{\alpha_0^{\langle\alpha\rangle} \mid \alpha_0 \in \mathscr{A}\}$. Conversely, we define the alphabet of all strings *not* tagged by a symbol $\mathscr{A}^{\bar{\alpha}} = \{\alpha' \in \mathscr{A} \mid \text{there is no } \alpha_0 \text{ such that } \alpha' = \alpha_0^{\langle\alpha\rangle}\}$.

Increment. The lexical operation $- \uparrow$ is defined as follows:

$$\alpha \uparrow = \begin{cases} \alpha_0^{\langle n+1\rangle} & \text{if } \alpha = \alpha_0^{\langle n\rangle},\ n \in \mathbb{N}, \\ \alpha & \text{otherwise.} \end{cases}$$

We define the alphabet $\mathscr{A} \uparrow = \{\alpha \uparrow \mid \alpha \in \mathscr{A}\}$.

Decrement. The lexical operation $- \downarrow$ is defined as follows:

$$\alpha \downarrow = \begin{cases} \alpha_0^{\langle n-1\rangle} & \text{if } \alpha = \alpha_0^{\langle n\rangle},\ n \in \mathbb{N},\ n > 0, \\ \alpha & \text{otherwise.} \end{cases}$$

We define the alphabet $\mathscr{A} \downarrow = \{\alpha \downarrow \mid \alpha \in \mathscr{A}\}$.

If a symbol is tagged more than once we will write the tags as follows:

$$(\alpha_0^{\langle\alpha_1\rangle})^{\langle\alpha_2\rangle} = \alpha_0^{\langle\alpha_1\alpha_2\rangle}.$$

Definition 4 (Extended regular expressions). The sets $\mathscr{R}_{\mathscr{A}}$ of extended regular expressions over finite alphabets $\mathscr{A}$ are defined inductively as the smallest sets for which:
Constants. $\emptyset, \varepsilon \in \mathscr{R}_{\mathscr{A}}$; if $\alpha \in \mathscr{A}$ then $\alpha \in \mathscr{R}_{\mathscr{A}}$.
Concatenation. If $R, R' \in \mathscr{R}_{\mathscr{A}}$ then $R \cdot R' \in \mathscr{R}_{\mathscr{A}}$.
Iteration. If $R \in \mathscr{R}_{\mathscr{A}}$, then $R^* \in \mathscr{R}_{\mathscr{A}}$.
Set operators. If $R, R' \in \mathscr{R}_{\mathscr{A}}$ then $R + R', R \cap R' \in \mathscr{R}_{\mathscr{A}}$.
Restriction. If $R \in \mathscr{R}_{\mathscr{A}}$, $\mathscr{A}' \subseteq \mathscr{A}$ then $R \upharpoonright \mathscr{A}' \in \mathscr{R}_{\mathscr{A}'}$.
Substitution. If $R, R' \in \mathscr{R}_{\mathscr{A}}$, $\omega \in \mathscr{A}^*$ then $R[\omega/R'] \in \mathscr{R}_{\mathscr{A}}$.
Tagging. If $R \in \mathscr{R}_{\mathscr{A}}$, $\alpha \in \mathscr{A}$ then $R^{\langle\alpha\rangle} \in \mathscr{R}_{\mathscr{A}\langle\alpha\rangle}$.
Increment/decrement. If $R \in \mathscr{R}_{\mathscr{A}}$, then $R\uparrow \in \mathscr{R}_{\mathscr{A}\uparrow}$, $R\downarrow \in \mathscr{R}_{\mathscr{A}\downarrow}$.
Shuffle. If $R, R' \in \mathscr{R}_{\mathscr{A}}$ then $R \bowtie R' \in \mathscr{R}_{\mathscr{A}}$.

If $\mathscr{A}$ is a finite alphabet, so are $\mathscr{A}^{\langle\alpha\rangle}$, $\mathscr{A}\uparrow$, $\mathscr{A}\downarrow$.

Constant $\emptyset$ denotes the empty language; constant ε denotes the empty string. The constant α is the language of the singleton sequence. Restriction is removing from all sequences in the language of a regular expression all symbols not in $\mathscr{A}'$.

The language of substitution $R[\omega/R']$ is the language of R where all occurrences of substring ω have been replaced by the strings of R'. If a finite number of substitutions must be performed simultaneously, we can write either $R[\omega_1/R'_1]\cdots[\omega_n/R'_n]$ or $R[\kappa]$ where κ is the finite function $(\omega_1 \mapsto R'_1, \ldots, \omega_n \mapsto R'_n)$.

The tagging of a language is the tagging of all symbols in its strings (similarly increment, decrement). The shuffle of two regular languages is defined as

Definition 5 (Shuffle). $\mathscr{L}_1 \bowtie \mathscr{L}_2 = \bigcup_{\omega_1 \in \mathscr{L}_1, \omega_2 \in \mathscr{L}_2} \omega_1 \bowtie \omega_2$, where $-\bowtie-$ is defined on strings as $\omega \bowtie \varepsilon = \varepsilon \bowtie \omega = \omega$ and $\alpha_1 \cdot \omega_1 \bowtie \alpha_2 \cdot \omega_2 = \alpha_1 \cdot (\omega_1 \bowtie \alpha_2 \cdot \omega_2) + \alpha_2 \cdot (\alpha_1 \cdot \omega_1 \bowtie \omega_2)$.

Proposition 6. *Every extended regular expression $R \in \mathscr{R}_{\mathscr{A}}$ denotes a regular language over $\mathscr{A}$.*

Proof. In addition to the normal regular expression operators $(\cdot, *, +)$, we have:
Set operators. Regular languages are closed under these set operations.
Restriction. From a finite-state automaton accepting R we can obtain the finite-state automaton accepting $R \upharpoonright \mathscr{A}'$ by replacing all transitions on inputs $\alpha \notin \mathscr{A}'$ with ε-transitions.
Substitution. The language of $R[\omega/R']$ is, by definition, the image of a regular-language homomorphism. It is known that regular languages are closed under homomorphisms.
Tagging, **increment**, **decrement**. Same reason as for substitution.
Shuffle. The shuffle operation has been studied quite extensively, [13] is a starting point in the literature. Shuffle is known to preserve regularity. □

We also need the notion of the *effective alphabet* of a regular expression, which is the set of all symbols appearing in the language denoted by that regular expression. The effective alphabet only depends on R.

Definition 7 (Effective alphabet). For any $R \in \mathscr{R}_{\mathscr{A}_0}$, the effective alphabet of R is $\lfloor R \rfloor = \{\alpha \in \mathscr{A}_0 \mid \omega \upharpoonright \alpha \neq \varepsilon \text{ for some } \omega \in R\}$.

A regular expression is *broadened* by shuffling with all strings *not* in its effective alphabet.

Definition 8 (Broadening). $\tilde{R} = R \bowtie (\mathscr{A} \setminus \lfloor R \rfloor)^*$.

The operation of broadening is relative to an alphabet $\mathscr{A}$, which must be appropriately specified in the context. Broadening will be used in modelling local variable declarations.

4.2. Interpretation of types

An alphabet $\mathscr{A}[\![-]\!]$ is associated with every data type τ and ground type σ; first-order types also have associated alphabets. The alphabets of types contain symbols $\mathsf{q} \in \mathsf{Q}[\![\theta]\!]$ called *questions*, and every question q has a set of *answers*, $\mathsf{a} \in \mathsf{A}_{\mathsf{q}}[\![\theta]\!]$.

Definition 9 (Type alphabets).

$$
\begin{aligned}
&\mathscr{A}[\![\mathbf{int}]\!] = \mathscr{Z} = \{-Z_{\max}, \ldots, -1, 0, 1, \ldots, Z_{\max}\} \subset \mathbb{Z},\\
&\mathscr{A}[\![\mathbf{bool}]\!] = \{tt, ff\},\\
&\mathsf{Q}[\![\mathbf{exp}\ \tau]\!] = \{q\}, \quad \mathsf{A}_q[\![\mathbf{exp}\ \tau]\!] = \mathscr{A}[\![\tau]\!],\\
&\mathsf{Q}[\![\mathbf{var}\ \tau]\!] = \{read\} \cup \{write(\alpha) \mid \alpha \in \mathscr{A}[\![\tau]\!]\},\\
&\quad \mathsf{A}_{read}[\![\mathbf{var}\ \tau]\!] = \mathscr{A}[\![\tau]\!], \quad \mathsf{A}_{write(\alpha)} = \{ok\},\\
&\mathsf{Q}[\![\mathbf{comm}]\!] = \{run\}, \quad \mathsf{A}_{run}[\![\mathbf{comm}]\!] = \{done\},\\
&\mathsf{Q}[\![\sigma_1 \to \cdots \to \sigma_k \to \sigma]\!] = \sum_{1 \leqslant i \leqslant k} \mathsf{Q}[\![\sigma_i]\!]^{\langle i \rangle} \cup \mathsf{Q}[\![\sigma]\!],\\
&\quad \mathsf{A}_{\mathsf{q}^{\langle i \rangle}}[\![\sigma_1 \to \cdots \to \sigma_k \to \sigma]\!] = (\mathsf{A}_{\mathsf{q}}[\![\sigma_i]\!])^{\langle i \rangle}, \quad \mathsf{q} \in \mathsf{Q}[\![\sigma_i]\!],\ 1 \leqslant i \leqslant k,\\
&\quad \mathsf{A}_{\mathsf{q}}[\![\sigma_1 \to \cdots \to \sigma_k \to \sigma]\!] = \mathsf{A}_{\mathsf{q}}[\![\sigma]\!], \quad \mathsf{q} \in \mathsf{Q}[\![\sigma]\!],\\
&\mathscr{A}[\![\theta]\!] = \mathsf{Q}[\![\theta]\!] \cup \bigcup_{\mathsf{q} \in \mathsf{Q}[\![\theta]\!]} \mathsf{A}_{\mathsf{q}}[\![\theta]\!].
\end{aligned}
$$

We use meta-variables α to range over any symbols, q over symbols which are questions and a over symbols which are answers.

Terms $\Gamma \vdash P : \theta$ are interpreted by an evaluation function $[\![-]\!]$ which maps them into a regular language. This regular language is defined over an alphabet induced by the environment:

Definition 10 (Environment alphabets).

$$
\begin{aligned}
\mathscr{A}[\![x : \theta]\!] &= \mathscr{A}[\![\theta]\!]^{\langle x \rangle},\\
\mathscr{A}[\![\Gamma]\!] &= \sum_{x : \theta \in \Gamma} \mathscr{A}[\![x : \theta]\!],\\
\mathscr{A}[\![\Gamma \vdash P : \theta]\!] &= \mathscr{A}[\![\Gamma]\!] \cup \mathscr{A}[\![\theta]\!].
\end{aligned}
$$

Example 11. $\mathscr{A}[\![f : \mathbf{comm} \to \mathbf{comm}]\!] = \{run^{\langle f\rangle}, done^{\langle f\rangle}, run^{\langle 1f\rangle}, done^{\langle 1f\rangle}\}$.

Every regular language that denotes the meaning of a term has a certain form, given by its possible initial and final moves. These moves indicate that a complete computation has occurred and give the result of the computation. In formulating the semantic definitions, it is useful to introduce the auxiliary notation $(\!| - |\!)$ defined below, taking advantage of the fact that for any type the regular language interpreting the term has a certain form. Exploiting this structure we can give more compact definitions.

Definition 12 (Semantic decompositions).

$$\begin{aligned}
[\![\Gamma \vdash C : \mathbf{comm}]\!] &= run \cdot (\!| \Gamma \vdash C : \mathbf{comm} |\!) \cdot done,\\
[\![\Gamma \vdash E : \mathbf{exp}\,\tau]\!] &= \sum_{\alpha \in [\![\tau]\!]} q \cdot (\!| \Gamma \vdash E : \mathbf{exp}\,\tau |\!)_{\alpha} \cdot \alpha,\\
[\![\Gamma \vdash V : \mathbf{var}\,\tau]\!] &= \sum_{\alpha \in [\![\tau]\!]} read \cdot (\!| \Gamma \vdash V : \mathbf{var}\,\tau |\!)_{\mathsf{r}\alpha} \cdot \alpha\\
&\quad + \sum_{\alpha \in [\![\tau]\!]} write(\alpha) \cdot (\!| \Gamma \vdash V : \mathbf{var}\,\tau |\!)_{\mathsf{w}\alpha} \cdot ok.
\end{aligned}$$

Intuitively, the language $(\!| C |\!)$ is the actual computation performed by C; $(\!| E |\!)_{\alpha}$ is only that particular computation of expression E which produces value α as a result. Variables contain two kinds of computations: $(\!| V |\!)_{\mathsf{r}\alpha}$, which happen when V reads value α, and $(\!| V |\!)_{\mathsf{w}\alpha}$ which happen when V writes value α. The full meaning of a term $[\![P]\!]$ is then the union of all these possible traces $(\!| P |\!)$. If it does not cause confusion, we may abbreviate the above notations to $[\![M : \theta]\!]$ or $[\![M]\!]$, and similarly for $(\!| - |\!)$.

The $(\!| - |\!)$ notation can in turn be defined in terms of the $[\![-]\!]$ notation. For example:

$$(\!| \Gamma \vdash C : \mathbf{comm} |\!) = \{w \mid run \cdot w \cdot done \in [\![\Gamma \vdash C : \mathbf{comm}]\!]\}.$$

The two notations are completely interchangeable, and we will use whichever is more convenient in context.

4.3. Expressions and control structures

The regular-language interpretation of integer and boolean constants is

Definition 13 (Constants).

$$\begin{aligned}
&(\!| n |\!)_{n} = \varepsilon, && (\!| n |\!)_{n'} = \emptyset,\ n' \neq n,\\
&(\!| \mathbf{true} |\!)_{\mathbf{true}} = \varepsilon, && (\!| \mathbf{true} |\!)_{\mathbf{false}} = \emptyset,\\
&(\!| \mathbf{false} |\!)_{\mathbf{false}} = \varepsilon, && (\!| \mathbf{false} |\!)_{\mathbf{true}} = \emptyset.
\end{aligned}$$

Definition 13 uses the property that the $[\![-]\!]$ and $(\!| - |\!)$ notations are equally expressive, because each can be formulated in terms of the other. The interpretations of the constants can also be expressed as

$$[\![n]\!] = q \cdot n, \quad [\![\mathbf{true}]\!] = q \cdot tt, \quad [\![\mathbf{false}]\!] = q \cdot ff.$$

The definitions of IA arithmetic-logic operators are

Definition 14 (Operators).

$$(\!|E_1 \bigstar E_2 : \mathbf{exp}\,\tau'|\!)_\alpha = \sum_{\substack{\alpha_1,\alpha_2 \in \mathscr{A}[\![\tau]\!] \\ \alpha = \alpha_1 \bigstar \alpha_2}} (\!|E_1 : \mathbf{exp}\,\tau|\!)_{\alpha_1} \cdot (\!|E_2 : \mathbf{exp}\,\tau|\!)_{\alpha_2}, \quad \alpha \in [\![\tau']\!].$$

Arithmetic operators over a finite set of integers can be interpreted in several ways. The first possibility is to have all operators *modulo* some maximum value, like in JAVA or C++. The second possibility is to leave the operators undefined if the value produced is out of range. This identifies the run-time error of numerical overflow with divergence, which is an expedient approximation, coarse but not entirely unacceptable (see for example, [3, Sections 2.7 and 5.1] for a discussion). A third possibility is to use the special values of **Infty**, $-$**Infty** and **NaN** (*not-a-number*) to denote positive respective negative overflow, or an indeterminate result. This approach is common in floating point operations (for example, ANSI/IEEE Standard 754-1985). All these approaches to handling overflow are compatible with the regular-language semantics, in that each can be modelled with appropriate changes to the semantic details.

Another semantic detail packaged in the definitions above is order of evaluation, left-to-right. Defining similar, but right-to-left, operators using this style of semantics can be done in the obvious way. Also, for logical operators, we have similar choices regarding lazy (*short-circuit*) or eager implementation of operators. Non-deterministic operators are also relatively easy to introduce. However, operators with parallel evaluation require substantial revisions of the semantic framework, which is unsurprising.

The imperative features are interpreted by

Definition 15 (Commands).

$$\begin{aligned}
&(\!|\mathbf{skip}|\!) = \varepsilon,\\
&(\!|\mathbf{diverge}|\!) = \emptyset,\\
&(\!|C; C'|\!) = (\!|C|\!) \cdot (\!|C'|\!),\\
&(\!|C; M|\!)_\alpha = (\!|C|\!) \cdot (\!|M|\!)_\alpha,\\
&(\!|\mathbf{while}\ B\ \mathbf{do}\ C|\!) = ((\!|B|\!)_{tt} \cdot (\!|C|\!))^* \cdot (\!|B|\!)_{ff},\\
&(\!|\mathbf{if}\ B\ \mathbf{then}\ C\ \mathbf{else}\ C'|\!) = (\!|B|\!)_{tt} \cdot (\!|C|\!) + (\!|B|\!)_{ff} \cdot (\!|C'|\!),
\end{aligned}$$

We can now consider a simple, standard example.

Example 16. $\Gamma \vdash \mathbf{while\ true\ do}\ C \equiv_{\mathbf{comm}} \mathbf{diverge}$.

Proof.

$$\begin{aligned}
[\![\mathbf{while\ true\ do}\ C]\!] &= run \cdot ((\!|\mathbf{true}|\!)_{tt} \cdot (\!|C|\!))^* \cdot (\!|\mathbf{true}|\!)_{ff} \cdot done\\
&= run \cdot (\varepsilon \cdot (\!|C|\!))^* \cdot \emptyset \cdot done\\
&= \emptyset = [\![\mathbf{diverge}]\!]. \qquad \square
\end{aligned}$$

The regular language semantics reveals some computational intuitions which are interesting in their own right. For example, **skip** is interpreted by the bracketing moves for commands enclosing the empty string. This suggests that it is a command which completes without having any effects. The regular expression interpreting any arithmetic-logic operator is decomposed into α-producing plays, where every such play is any concatenation of plays producing α_1 and α_2 in the arguments, if and only if $\alpha_1 \bigstar \alpha_2 = \alpha$. Composition of commands is simply concatenation of plays. Looping is interpreted as an iteration of plays in the guard of the loop producing true concatenated with complete plays in the body, followed by one single play in the guard, producing false. Remarkably, this is exactly the trace-based interpretation used to interpret iteration as early as the 1970s (see for example Section 2.3.4 in [11]). It is also similar to Brookes's trace-based interpretation of parallel IA [3]. Non-termination **diverge** is interpreted as the empty set of complete, i.e. terminating, plays.

4.4. Free identifiers and functions

Free identifiers, of ground and function type, are given a concrete interpretation; i.e., they are represented by a regular language, just like a closed term. This "flattening" of the semantics, so that no higher-order entities such as functions or quantifiers are needed in the model, is arguably the most remarkable feature of game semantics.

In order to interpret free identifiers, we use regular languages which represent the ubiquitous *copy-cat* strategies.

Definition 17 (Copy-cat). The copy-cat regular languages $\mathscr{K}^{\alpha}_{\theta}$ where α is an arbitrary symbol, are defined as

$$\mathscr{K}^{\alpha}_{\sigma_1\to\cdots\to\sigma_k} = \sum_{\mathsf{q}\in\mathsf{Q}[\![\sigma_k]\!]} \sum_{\mathsf{a}\in\mathsf{A}_{\mathsf{q}}[\![\sigma_k]\!]} \mathsf{q}\cdot\mathsf{q}^{\langle\alpha\rangle}\cdot\left(\sum_{1\leqslant j<k}(\mathscr{L}^{\alpha,j}_{\sigma_j})\right)^{*}\cdot\mathsf{a}^{\langle\alpha\rangle}\cdot\mathsf{a},$$

where $\mathscr{L}^{\alpha,j}_{\sigma} = \sum_{\mathsf{q}\in\mathsf{Q}[\![\sigma]\!]}\sum_{\mathsf{a}\in\mathsf{A}_{\mathsf{q}}[\![\sigma]\!]}\mathsf{q}^{\langle j\alpha\rangle}\cdot\mathsf{q}^{\langle j\rangle}\cdot\mathsf{a}^{\langle j\rangle}\cdot\mathsf{a}^{\langle j\alpha\rangle}$.

The languages $\mathscr{L}^{\alpha,j}_{\sigma}$ are traces representing a function using an argument; the languages $\mathscr{K}^{\alpha}_{\sigma_1\to\cdots\to\sigma_k}$ represent all the possible ways in which a function can use its arguments.

Then, the definition of free identifiers is

Definition 18 (Identifiers). $[\![\Gamma, x:\theta\vdash x:\theta]\!] = \mathscr{K}^{x}_{\theta}$.

Example 19.

$$\begin{aligned}&[\![f:\mathbf{comm}\to\mathbf{comm}\vdash f:\mathbf{comm}\to\mathbf{comm}]\!]\\&\quad= run\cdot run^{\langle f\rangle}\cdot(run^{\langle 1f\rangle}\cdot run^{\langle 1\rangle}\cdot done^{\langle 1\rangle}\cdot done^{\langle 1f\rangle})^{*}\cdot done^{\langle f\rangle}\cdot done\\&\quad= \mathscr{K}^{f}_{\mathbf{comm}\to\mathbf{comm}}.\end{aligned}$$

Conceptually, the moves tagged with f represent the effects of calling then returning from the function; moves tagged by $1f$ are the effects caused by f whenever it evaluates its first, and in this example its only, argument. The argument may be evaluated an arbitrary number of times, sequentially (no interleaving), hence the Kleene closure. The moves with only numerical tags correspond to the formal parameters.

Example 20.

$$\begin{aligned}
&[\![f : \mathbf{comm} \to \mathbf{comm} \to \mathbf{comm} \vdash f : \mathbf{comm} \to \mathbf{comm} \to \mathbf{comm}]\!] \\
&\quad = \mathscr{K}^{f}_{\mathbf{comm}\to\mathbf{comm}\to\mathbf{comm}} \\
&\quad = run \cdot run^{\langle f\rangle} \cdot (run^{\langle 1f\rangle} \cdot run^{\langle 1\rangle} \cdot done^{\langle 1\rangle} \cdot done^{\langle 1f\rangle} \\
&\qquad + run^{\langle 2f\rangle} \cdot run^{\langle 2\rangle} \cdot done^{\langle 2\rangle} \cdot done^{\langle 2f\rangle})^{*} \cdot done^{\langle f\rangle} \cdot done.
\end{aligned}$$

The example above illustrates why the games model gives an intuitively appealing account of sequentiality, because it makes obvious the property that function f can only evaluate one of its arguments after it has completed the evaluation of the other argument.

Abstraction is interpreted as a re-tagging of symbols in the language. Conceptually, this corresponds to the "moving" of the identifier from the environment to the term. This rule is another instance of the remarkable "flatness" and concreteness of the game semantics.

Definition 21 (Abstraction).

$$[\![\Gamma \vdash \lambda m{:}\,\sigma.P : \sigma \to \theta]\!] = ([\![\Gamma, m : \sigma \vdash P : \theta]\!] \uparrow)[\kappa],$$

where $\kappa(\alpha^{\langle m\rangle}) = \alpha^{\langle 1\rangle}$.

The moves associated with m become "anonymous", and are tagged with a number. In order to keep the tags unique, all other symbols are incremented.

Application is modelled by trace-level substitution:

Definition 22 (Application).

$$[\![PM : \theta]\!] = ([\![P : \sigma \to \theta]\!][\kappa]) \downarrow,$$

where $\kappa(\mathsf{q}^{\langle 1\rangle} \cdot \mathsf{a}^{\langle 1\rangle}) = \{w \mid \mathsf{q} \cdot w \cdot \mathsf{a} \in [\![M : \sigma]\!]\}$, for any $\mathsf{q} \in \mathsf{Q}[\![\sigma]\!]$, $\mathsf{a} \in \mathsf{A}_{\mathsf{Q}}[\![\sigma]\!]$.

The moves corresponding to the outermost identifier bound by lambda are tagged with 1, so upon application the pairs of symbols corresponding to the formal parameter are substituted by the concrete traces of the argument. The rest of the indices are decremented. This mechanism is quite similar to the representation of lambda calculus using de Bruijn indices [4].

Example 23.

$$
\begin{aligned}
&[\![(\lambda x\colon \mathbf{expint}.x+1)7]\!]\\
&\quad = ([\![\lambda x\colon \mathbf{expint}.x+1]\!][\kappa])\downarrow\\
&\quad = ((([\![x : \mathbf{expint} \vdash x+1]\!])\uparrow[\kappa'])[\kappa])\downarrow\\
&\quad = \left(\left(\left(\sum_{n,n'\in\mathscr{Z}} q\cdot(\!|x : \mathbf{expint}\vdash x|\!)\cdot(\!|1|\!)_{n'}\cdot(n+n')\right)\uparrow[\kappa']\right)[\kappa]\right)\downarrow\\
&\quad = \left(\left(\left(\sum_{n\in\mathscr{Z}} q\cdot q^{\langle x\rangle}\cdot n^{\langle x\rangle}\cdot(n+1)\right)\uparrow[\kappa']\right)[\kappa]\right)\downarrow\\
&\quad = \left(\left(\left(\sum_{n\in\mathscr{Z}} q\cdot q^{\langle x\rangle}\cdot n^{\langle x\rangle}\cdot(n+1)\right)[\kappa']\right)[\kappa]\right)\downarrow\\
&\quad = \left(\left(\sum_{n\in\mathscr{Z}} q\cdot q^{\langle 1\rangle}\cdot n^{\langle 1\rangle}\cdot(n+1)\right)[\kappa]\right)\downarrow\\
&\quad = \left(\left(\sum_{n\in\mathscr{Z}} q\cdot q^{\langle 1\rangle}\cdot n^{\langle 1\rangle}\cdot(n+1)\right)[\kappa]\right)\downarrow\\
&\quad = (q\cdot 8)\downarrow\\
&\quad = q\cdot 8,
\end{aligned}
$$

where $\kappa'(q^{\langle x\rangle}\cdot n^{\langle x\rangle}) = q^{\langle 1\rangle}\cdot n^{\langle 1\rangle}$ and $\kappa(q^{\langle 1\rangle}\cdot n^{\langle 1\rangle}) = \begin{cases}\varepsilon & \text{if } n=7\\ \emptyset & \text{if } n\neq 7.\end{cases}$

4.5. Store

Reading and writing to a variable is achieved by dereferencing and assignment, respectively.

Definition 24 (Variable manipulation).

$$
\begin{aligned}
&(\!|\Gamma\vdash !V : \mathbf{exp}\,\tau|\!)_\alpha = (\!|\Gamma\vdash V : \mathbf{var}\,\tau|\!)_{\mathsf{r}\alpha}, \quad \alpha\in\mathscr{A}[\![\tau]\!]\\
&(\!|\Gamma\vdash V := E : \mathbf{comm}|\!) = \sum_{\alpha\in\mathscr{A}[\![\tau]\!]} (\!|\Gamma\vdash E : \mathbf{exp}\,\tau|\!)_\alpha\cdot(\!|\Gamma\vdash V : \mathbf{var}\,\tau|\!)_{\mathsf{w}\alpha}.
\end{aligned}
$$

Note that the semantics above imposes no causal correlation between the reads and writes of variables. For example, the expression with side-effects $v := 1; !v$ has the interpretation:

$$
[\![v : \mathbf{varint}\vdash v := 1; !v]\!] = \sum_{n\in\mathscr{Z}} q\cdot write(1)^{\langle v\rangle}\cdot ok^{\langle v\rangle}\cdot read^{\langle v\rangle}\cdot n^{\langle v\rangle}\cdot n.
$$

In other words, upon writing 1 to the variable it is still possible to get any value when reading the variable. Why is this possible? The reason is that the identifier v may be bound by function application to *any* variable-typed term, for example, $(\lambda v\colon \mathbf{varint}.v := 1; !v)(\mathbf{if}\ !x = 1\ \mathbf{then}\ x := 7; x\ \mathbf{else}\ x := 0; x)$. Even worse than in the case of side-effect

free IA, variables are not even guaranteed to return the same value upon consecutive readings. The reason is the same, the variable identifier may be bound to a phrase that has side-effects which may include changing the variable itself.

Only variables that are known to be locally declared in the evaluation context are guaranteed to be "well-behaved" in the sense that there is an expected causal connection between the values that are read and the values that are written. This property is captured by the following regular language:

Definition 25 (Variable stability).

$$\gamma^{v}_{\mathbf{var}\,\tau} = (read^{\langle v\rangle} \cdot \alpha_{\tau}^{\langle v\rangle})^* \cdot \left(\sum_{\alpha \in \mathscr{A}[\![\tau]\!]} write(\alpha)^{\langle v\rangle} \cdot ok^{\langle v\rangle} \cdot (read^{\langle v\rangle} \cdot \alpha^{\langle v\rangle})^* \right)^*,$$

where $\alpha_{\mathbf{int}} = 0$, $\alpha_{\mathbf{bool}} = \mathbf{false}$.

Initially, the value read from the variable v is the default value α_τ. Any legal sequence consists of a *write* followed by an arbitrary number of *read*s, all yielding the value that was written. In the terminology used by Reynolds, a stable variable is a variable which is *good* and *not interfered with*.

The semantics of the local-variable block consists of two operations: imposing the stable-variable behaviour and then removing all occurrences of actions of that variable as it becomes invisible outside its binding scope:

Definition 26 (Block variable).

$$[\![\Gamma \vdash \mathbf{new}\ \tau\ v\ \mathbf{in}\ M : \sigma]\!] = ([\![\Gamma, v : \mathbf{var}\,\tau \vdash M : \sigma]\!] \cap \widetilde{\gamma^{v}_{\mathbf{var}\,\tau}}) \upharpoonright \mathscr{A}^{\bar{v}},$$

where $\mathscr{A} = \mathscr{A}[\![\Gamma, v : \mathbf{var}\,\tau \vdash M : \sigma]\!]$.

The actions not tagged by v are not constrained. All other actions are not constrained by stability, and they depend on the context. They are introduced at the point of the definition of the variable in a block using the broadening operation (Definition 8). Scope is modeled by restriction, which hides away all interactions of v (Definition 4). The following section contains numerous examples showing this definition at work.

Terms of the language fragment interpreted in this section are observationally equivalent if and only if their regular language interpretations are equal.

Theorem 27 (Full abstraction). *For any two terms of the second-order fragment of IA, $\Gamma \vdash M \equiv_\sigma N$ iff $[\![\Gamma \vdash M]\!] = [\![\Gamma \vdash N]\!]$.*

The proof is given in Section 6. Using Theorem 27 we can prove numerous interesting example equivalences.

5. Examples of equational reasoning

We have presented so far a substantial part of the semantics of second-order IA. The part that is missing, introduced later in Section 7, is function definition. Although the language fragment presented so far is not complete, it contains all the definitions necessary to prove many interesting example equivalences.

5.1. Locality

$$c : \mathbf{comm} \vdash \mathbf{newint}\, v \ \mathbf{in}\ c \equiv_{\mathbf{comm}} c.$$

This deceptively simple equivalence ([16, Example 1]) is not validated by the traditional models of imperative computation relying on a global store model, traceable back to Scott and Strachey [33]. It reflects the fact that a non-locally defined procedure cannot modify a local variable. It was first proved in the "possible worlds" model of Reynolds and Oles, constructed using functor categories [24].

Proof.

$$\begin{aligned}
&[\![c : \mathbf{comm} \vdash \mathbf{newint}\, v \ \mathbf{in}\ c : \mathbf{comm}]\!] \\
&\quad = ([\![c : \mathbf{comm}, v : \mathbf{var}\,\tau \vdash c : \mathbf{comm}]\!] \cap \widetilde{\gamma^v_\tau}) \upharpoonright \mathscr{A}^{\bar{v}} \\
&\quad = (run \cdot run^{\langle c\rangle} \cdot done^{\langle c\rangle} \cdot done \cap \widetilde{\gamma^v_\tau}) \upharpoonright \mathscr{A}^{\bar{v}} \\
&\quad = run \cdot run^{\langle c\rangle} \cdot done^{\langle c\rangle} \cdot done \\
&\qquad \text{because } run \cdot run^{\langle c\rangle} \cdot done^{\langle c\rangle} \cdot done \in (\mathscr{A}^{\bar{v}})^* \\
&\quad = [\![c : \mathbf{comm} \vdash c : \mathbf{comm}]\!]. \qquad \square
\end{aligned}$$

5.2. Snapback

$$\begin{aligned}
f : \mathbf{comm} \to \mathbf{comm} \vdash\ & \\
& \mathbf{newint}\, v \ \mathbf{in} \\
& \quad v := 0; f(v := 1); \qquad\qquad \equiv_{\mathbf{comm}} f(\mathbf{diverge}). \\
& \quad \mathbf{if}\ !v = 1\ \mathbf{then\ diverge\ else\ skip}
\end{aligned}$$

This example [20, Section 7.1] captures the intuition that state changes are in some way irreversible. A procedure executing an argument which is a command changes the state in a way that cannot be undone from within the procedure. If procedure f uses its argument both sides will fail to terminate; if procedure f does not use its argument, the behaviour of each side will be identical because of the locality of v, as seen above.

The first model to correctly address this issue was O'Hearn and Reynolds's interpretation of IA using the polymorphic linear lambda calculus [20]. Reddy also addressed this issue using a novel "object semantics" approach [27], but in a particular flavour of IA known as interference-controlled ALGOL [18]. A further development of the model, which also satisfies this equivalence, is O'Hearn and Reddy's [19], a model fully abstract for the second-order subset.

Proof. We proceed in a "bottom up" fashion. The following evaluation is routine:

$$\begin{aligned}
&[\![v : \mathbf{varint}, f : \mathbf{comm} \to \mathbf{comm} \vdash f(v := 1) : \mathbf{comm}]\!] \\
&\quad = (run \cdot run^{\langle f \rangle} \cdot (run^{\langle 1f \rangle} \cdot run^{\langle 1 \rangle} \cdot done^{\langle 1 \rangle} \cdot done^{\langle 1f \rangle})^* \cdot done^{\langle f \rangle} \cdot done) \\
&\qquad [run^{\langle 1 \rangle} \cdot done^{\langle 1 \rangle} / write(1)^{\langle v \rangle} \cdot ok^{\langle v \rangle}] \\
&\quad = run \cdot run^{\langle f \rangle} \cdot (run^{\langle 1f \rangle} \cdot write(1)^{\langle v \rangle} \cdot ok^{\langle v \rangle} \cdot done^{\langle 1f \rangle})^* \cdot done^{\langle f \rangle} \cdot done.
\end{aligned}$$

The following evaluation is also routine:

$$[\![\mathbf{if}\ !v = 1\ \mathbf{then\ diverge\ else\ skip} : \mathbf{comm}]\!] = \sum_{1 \neq \alpha \in \mathscr{Z}} run \cdot read^{\langle v \rangle} \cdot \alpha^{\langle v \rangle} \cdot done.$$

Using the two above, we have that

$$\begin{aligned}
&[\![v := 0; f(v := 1); \mathbf{if}\ !v = 1\ \mathbf{then\ diverge\ else\ skip} : \mathbf{comm}]\!] \\
&\quad = run \cdot write(0)^{\langle v \rangle} \cdot ok^{\langle v \rangle} \cdot run^{\langle f \rangle} \cdot (run^{\langle 1f \rangle} \cdot write(1)^{\langle v \rangle} \cdot ok^{\langle v \rangle} \cdot done^{\langle 1f \rangle})^* \\
&\qquad \cdot done^{\langle f \rangle} \cdot \sum_{1 \neq \alpha \in \mathscr{Z}} read^{\langle v \rangle} \cdot \alpha^{\langle v \rangle} \cdot done.
\end{aligned}$$

The first part of the interpretation of v as a block variable is the intersection with $\widetilde{\gamma^{v}_{\mathbf{int}}}$. We note that:

- if the iteration $(*)$ is empty then the stability of v forces all the subsequent *read*s to produce 0;
- if the iteration is non-empty then the stability of v forces the subsequent *read*s to produce 1; but the condition $\alpha \neq 1$ stipulates that 1 cannot be produced. Therefore, the entire trace is empty in this case.

So,

$$\begin{aligned}
&run \cdot write(0)^{\langle v \rangle} \cdot ok^{\langle v \rangle} \cdot run^{\langle f \rangle} \cdot (run^{\langle 1f \rangle} \cdot write(1)^{\langle v \rangle} \cdot ok^{\langle v \rangle} \cdot done^{\langle 1f \rangle})^* \\
&\quad \cdot done^{\langle f \rangle} \cdot \sum_{1 \neq \alpha \in \mathscr{Z}} read^{\langle v \rangle} \cdot \alpha^{\langle v \rangle} \cdot done \cap \widetilde{\gamma^{\langle v \rangle}_{\mathbf{int}}} \\
&= run \cdot write(0)^{\langle v \rangle} \cdot ok^{\langle v \rangle} \cdot run^{\langle f \rangle} \cdot done^{\langle f \rangle} \cdot read^{\langle v \rangle} \cdot 0^{\langle v \rangle} \cdot done.
\end{aligned}$$

After restriction to $\mathscr{A}^{\bar{v}}$, we have that

$$[\![LHS]\!] = run \cdot run^{\langle f \rangle} \cdot done^{\langle f \rangle} \cdot done.$$

It can be immediately seen that this is also the interpretation of the right-hand side. □

5.3. Invariant preservation

$$f : \mathbf{comm} \to \mathbf{comm} \vdash \begin{array}{l} \mathbf{newint}\ v\ \mathbf{in} \\ \quad v := 0; f(v := !v + 2); \\ \quad \mathbf{if}\ !v\ \mathbf{mod}\ 2 = 0\ \mathbf{then\ diverge\ else\ skip} \end{array} \equiv \mathbf{diverge}.$$

The principle illustrated in this example [16, Example 5] is that of invariant preservation. Although procedure f has read and write access to variable v, the access is only through command $v :=!v + 2$, so the variable can only be incremented by two. Therefore, variable v will always hold an even value.

Proof. Following the same bottom-up approach, it is routine to evaluate

$$\left[\!\left[f : \mathbf{comm} \to \mathbf{comm} \vdash \begin{array}{l} v := 0; f(v :=!v + 2); \\ \mathbf{if}\ !v\ \mathbf{mod}\ 2 = 0\ \mathbf{then\ diverge\ else\ skip} : \mathbf{comm} \end{array} \right]\!\right]$$

to

$$\begin{aligned} &run \cdot write(0)^{\langle v\rangle} \cdot ok^{\langle v\rangle} \cdot run^{\langle f\rangle} \cdot \left(\sum_{\alpha\in\mathscr{Z}} run^{\langle 1 f\rangle} \cdot read^{\langle v\rangle} \cdot \alpha^{\langle v\rangle} \right. \\ &\quad \left. \cdot write(\alpha + 2)^{\langle v\rangle} \cdot ok^{\langle v\rangle} \cdot done^{\langle 1 f\rangle} \right)^{*} \cdot done^{\langle f\rangle} \cdot \left(\sum_{\alpha\in\mathscr{Z},\alpha \bmod 2=1} read^{\langle v\rangle} \cdot \alpha^{\langle v\rangle} \right) \\ &\quad \cdot done. \end{aligned}$$

It is immediately seen that upon introducing the stable-variable constraint for v, this regular expression becomes $\emptyset$, because for any $k \geqslant 0$ iterations the value stored in v is even, equal to $2 \times k$. This contradicts the clause $\alpha \bmod 2 = 1$ in the second part of the trace.

Therefore, $[\![LHS]\!] = \emptyset = [\![\mathbf{diverge}]\!]$. □

5.4. Representation independence

$$f : \mathbf{comm} \to \mathbf{expbool} \to \mathbf{comm} \vdash \begin{array}{l} \mathbf{newint}\ v\ \mathbf{in} \\ \quad v := 0; \\ \quad f(v := 1)(!v = 0) \end{array} \equiv_{\mathbf{comm}} \begin{array}{l} \mathbf{newint}\ v\ \mathbf{in} \\ \quad v := 0; \\ \quad f(v := -1)(!v = 0). \end{array}$$

The two sides of the equivalence represent two possible implementations of a switch object. The switch is initially in the *on* state. The first argument is a method that changes the state of the switch to *off*; the second one returns a boolean expression representing the state. The first implementation changes the state by assigning one, the second by assigning negative one. The behaviour of the two implementations should, however, be identical, illustrating a principle of representation independence. Although Oles does not prove this example, it can be proved using his possible-worlds model [23].

Proof. As in the previous examples, evaluating the phrase bottom-up is mechanical:

$$[\![f : \mathbf{comm} \to \mathbf{expbool} \to \mathbf{comm} \vdash v := 0; f(v := 1)(!v = 0) : \mathbf{comm}]\!]$$

is given by

$$run \cdot write(0)^{\langle v\rangle} \cdot ok^{\langle v\rangle} \cdot run^{\langle f\rangle} \cdot \Bigg(\overbrace{run^{\langle 1f\rangle} \cdot write(1)^{\langle v\rangle} \cdot ok^{\langle v\rangle} \cdot done^{\langle 1f\rangle}}^{\mathscr{R}_{\mathrm{write}}} + \underbrace{q^{\langle 2f\rangle} \cdot read^{\langle v\rangle} \cdot 0^{\langle v\rangle} \cdot tt^{\langle 2f\rangle}}_{\mathscr{R}_{tt}} + \underbrace{\sum_{0\neq\alpha\in\mathscr{Z}} q^{\langle 2f\rangle} \cdot read^{\langle v\rangle} \cdot \alpha^{\langle v\rangle} \cdot ff^{\langle 2f\rangle}}_{\mathscr{R}_{ff}} \Bigg)^* \cdot done^{\langle f\rangle} \cdot done.$$

When the stability property for v is imposed, it is obvious that in the iterated part, R_{tt} must occur only before R_{write}, and R_{ff} only after

$$\widetilde{\gamma^v_{\mathbf{int}}} \cap \Bigg(run \cdot write(0)^{\langle v\rangle} \cdot ok^{\langle v\rangle} \cdot run^{\langle f\rangle} \cdot \Bigg(\overbrace{run^{\langle 1f\rangle} \cdot write(1)^{\langle v\rangle} \cdot ok^{\langle v\rangle} \cdot done^{\langle 1f\rangle}}^{\mathscr{R}_{\mathrm{write}}} + \underbrace{q^{\langle 2f\rangle} \cdot read^{\langle v\rangle} \cdot 0^{\langle v\rangle} \cdot tt^{\langle 2f\rangle}}_{\mathscr{R}_{tt}} + \underbrace{\sum_{0\neq\alpha\in\mathscr{Z}} q^{\langle 2f\rangle} \cdot read^{\langle v\rangle} \cdot \alpha^{\langle v\rangle} \cdot ff^{\langle 2f\rangle}}_{\mathscr{R}_{ff}} \Bigg)^* \cdot done^{\langle f\rangle} \cdot done \Bigg)$$

$$= run \cdot write(0)^{\langle v\rangle} \cdot ok^{\langle v\rangle} \cdot run^{\langle f\rangle} \cdot (\mathscr{R}_{tt}^* \cdot \mathscr{R}_{\mathrm{write}}^* \cdot (\varepsilon + \mathscr{R}_{\mathrm{write}} \cdot (\mathscr{R}_{\mathrm{write}} + \mathscr{R}_{ff})^*)) \cdot done^{\langle f\rangle} \cdot done.$$

Restriction to $\mathscr{A}^{\bar{v}}$ gives $[\![LHS]\!]$:

$$run \cdot run^{\langle f\rangle} \cdot ((q^{\langle 2f\rangle} \cdot tt^{\langle 2f\rangle})^* \cdot (run^{\langle 1f\rangle} \cdot done^{\langle 1f\rangle})^* \cdot (\varepsilon + run^{\langle 1f\rangle} \cdot done^{\langle 1f\rangle} \cdot (run^{\langle 1f\rangle} \cdot done^{\langle 1f\rangle} + q^{\langle 2f\rangle} \cdot ff^{\langle 2f\rangle})^*)) \cdot done^{\langle f\rangle} \cdot done.$$

Note that this interpretation captures the dynamics of the term quite well. In English, it shows that in function f the second argument will always evaluate to true, until the first argument is evaluated; thereafter, the second argument will evaluate to false, regardless of whether the first argument is evaluated or not.

Evaluating RHS gives the same regular expression. □

5.5. Parametricity

$$f : \mathbf{comm} \to \mathbf{comm} \vdash \mathbf{newint}\, v \ \mathbf{in}\ f(v :=\!!v + 1) \equiv_{\mathbf{comm}} f(\mathbf{skip}).$$

This example seems similar to the locality or representation-independence examples. However, the fact that the expression on the left-hand side changes the state each time f calls its argument, whereas the expression on the right-hand side has no effect on the state, leads to technical problems in models with explicit store. Several such examples are given by O'Hearn and Tennent [21], who deal with them using a model constructed using a relation-preserving functor category.

Proof. Let us assume that we deal with overflow using special values or "wrap-around" arithmetic.

We will impose the local-variable constraint on

$$\begin{aligned}&[\![f : \mathbf{comm} \to \mathbf{comm} \vdash f(v :=!v+1) : \mathbf{comm}]\!]\\ &\quad = run \cdot run^{\langle f\rangle} \cdot \left(\sum_{\alpha\in\mathscr{Z}} run^{\langle 1f\rangle} \cdot read^{\langle v\rangle} \cdot \alpha^{\langle v\rangle} write(\alpha+1)^{\langle v\rangle} \cdot ok^{\langle v\rangle} \cdot done^{\langle 1f\rangle}\right)^*\\ &\qquad \cdot done^{\langle f\rangle} \cdot done.\end{aligned}$$

After imposing the local variable constraint, the result would become a set of traces of the form:

$$\begin{aligned}run \cdot run^{\langle f\rangle} &\cdot run^{\langle 1f\rangle} \cdot read^{\langle v\rangle} \cdot 0^{\langle v\rangle} \cdot write(1)^{\langle v\rangle} \cdot ok^{\langle v\rangle} \cdot done^{\langle 1f\rangle}\\ &\cdot run^{\langle 1f\rangle} \cdot read^{\langle v\rangle} \cdot 1^{\langle v\rangle} \cdot write(2)^{\langle v\rangle} \cdot ok^{\langle v\rangle} \cdot done^{\langle 1f\rangle}\\ &\cdot run^{\langle 1f\rangle} \cdot read^{\langle v\rangle} \cdot 2^{\langle v\rangle} \cdot write(3)^{\langle v\rangle} \cdot ok^{\langle v\rangle} \cdot done^{\langle 1f\rangle} \cdot \cdots \cdot done^{\langle f\rangle}\\ &\cdot done.\end{aligned}$$

After restriction to $\mathscr{A}^{\bar{v}}$:

$$[\![LHS]\!] = run \cdot run^{\langle f\rangle} \cdot (run^{\langle 1f\rangle} \cdot done^{\langle 1f\rangle})^* \cdot done^{\langle f\rangle} \cdot done = [\![RHS]\!],$$

so the equivalence stands. □

Note that here the result depends on how addition is implemented over a finite set! If overflow is interpreted as divergence then the meaning of the LHS is

$$run \cdot run^{\langle f\rangle} \cdot \left(\sum_{k\leqslant N_{\max}} \underbrace{run^{\langle 1f\rangle} \cdot done^{\langle 1f\rangle} \cdots run^{\langle 1f\rangle} \cdot done^{\langle 1f\rangle}}_{k \text{ times}}\right) \cdot done^{\langle f\rangle} \cdot done,$$

which is *not* the same as $[\![RHS]\!]$. So if overflow leads to abortion then the equivalence actually fails!

5.6. Mechanical verification

As one can see in Example 5.4, proofs using regular expressions can be quite complex. We have implemented a prototype tool based on the model presented here which compiles IA terms into their regular-language interpretations, represented as finite-state machines. For both terms in Example 5.4, the tool generates the finite-state machine displayed in Fig. 5.

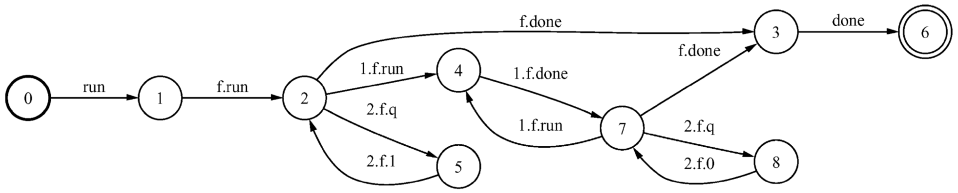


Fig. 5. Example 5.4.

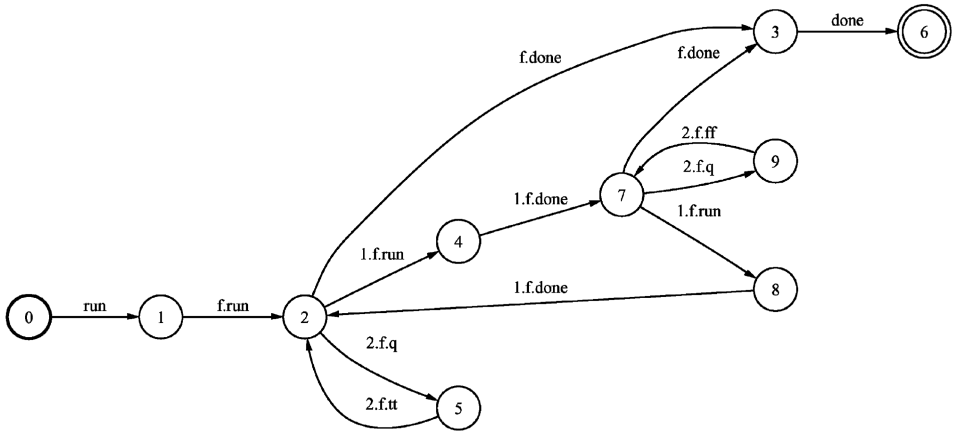


Fig. 6. An on–off switch.

A variant of Example 5.4 is a switch with both *on* and *off* capabilities, as presented in [5]. We want to see that two alternative implementations of the switch are equivalent:

$$f : \mathbf{comm} \to \mathbf{expbool} \to \mathbf{comm} \vdash$$

$$\begin{array}{l} \mathbf{newint}\ v\ \mathbf{in} \\ \quad v := 1; \\ \quad f(v := -!v)(!v > 0) \end{array} \equiv_{\mathbf{comm}} \begin{array}{l} \mathbf{newbool}\ v\ \mathbf{in} \\ \quad v := \mathbf{true}; \\ \quad f(v := \mathbf{not}\ !v)(!v). \end{array}$$

Our tool does indeed show that both implementations have the same regular language interpretations. The corresponding finite-state machine is shown in Fig. 6.

6. Correctness of the regular-language semantics

In this section and the next we will show that the regular-language semantics is a fully abstract model for second-order IA. This is the case because our model is isomorphic to the fully abstract Abramsky–McCusker game model.

The material in this section necessarily refers to technical details of the games model, which we do not reproduce here. The reader unfamiliar with these details should refer to the original papers. (*loc. cit.*)

We begin by observing that, for the types we are considering, the justification pointers in the game semantics are redundant.

Lemma 28. *Let A be a game interpreting a second-order type of IA, and s be any legal play of A, in which both players are subject to the bracketing and visibility conditions. If s has only one initial move, then it is completely determined by its underlying sequence of moves.*

Proof. The game A can be written as

$$(B_{0,0} \to \cdots \to B_{0,l_0} \to B_0) \to \cdots \to (B_{i,0} \to \cdots \to B_{i,l_i} \to B_i) \to B,$$

where the $B_{i,j}, B_i$ and B are base-type games. We shall show that any move either player can make is either initial, so requires no justifier, or has exactly one possible justifying move.

For answer-moves, this is immediate from the bracketing condition, so we need only consider question-moves. For initial moves, there is nothing to prove. There are two kinds of non-initial question-moves

- P-moves in one of the B_j,
- O-moves in one of the $B_{j,k}$.

P-moves in the B_j are justified by the unique initial move. For an O-move in $B_{j,k}$, the justifier is a P-question in B_j. Any such move is itself justified by the unique initial move, so in the O-view of the position, the *most recent* P-question in B_j is also the *only* P-question in B_j, and by the visibility condition, this must be the justifier of the O-move we are considering. □

In light of this lemma, we will ignore justification pointers in all games from now on.

In the following, we will use $[\![M]\!]$ to denote the regular-language interpretation of a term M and $[\![M]\!]^{\mathsf{comp}}$ to denote the set of complete plays in the strategy interpreting M in the games model, with justification pointers deleted.

For any type $\theta = \sigma_1 \to \cdots \to \sigma_k \to \sigma$ of second-order IA, the set of legal plays of its game representation is isomorphic to a regular language. Let us call this regular language $\mathscr{R}[\![\theta]\!]$. The isomorphism ρ_0 consists of the tagging of all moves in the game model of σ_j with j.

Lemma 29 (Type representation). $\mathsf{P}_\theta^{\mathsf{comp}} \overset{\rho_0}{\cong} \mathscr{R}[\![\theta]\!]$, *where by* $\mathsf{P}_\theta^{\mathsf{comp}}$ *we denote the set of* complete plays *in the game interpreting* σ.

Proof (by induction on the structure of θ).

Ground types are interpreted by finite sets of complete plays, which are by definition regular. Let $\mathscr{R}[\![\sigma]\!] = \mathsf{P}_\sigma^{\mathsf{comp}}$.

First-order types of the form $\sigma_1 \to \cdots \to \sigma_k \to \sigma$ have sets of complete plays isomorphic to the regular language

$$\mathscr{R}[\![\sigma_1 \to \cdots \to \sigma_k \to \sigma]\!] = \sum_{\mathsf{q},\mathsf{a}} \mathsf{q} \cdot \left(\sum_{1 \leqslant j \leqslant k} \mathscr{R}[\![\sigma_j]\!]^{\langle j \rangle} \right)^* \cdot \mathsf{a}, \tag{1}$$

where $\mathsf{q}, \mathsf{a} \in \mathsf{M}_\sigma$, such that q is a question and a is an answer. This is because the opening question must be in σ, followed by an arbitrary number of repetitions of a complete set of plays in any σ_j. Moves of σ_j are tagged with j as part of the disjoint summing of the moves of the games; this is consistent with the definition of ρ_0. □

For any term $x_0 : \theta_0, \ldots, x_k : \theta_k \vdash M : \sigma_1 \to \cdots \to \sigma_l \to \sigma$, let isomorphism ρ_1, from sets of moves to alphabets, be:

- the unique tagging of the moves in θ_j with x_j and, additionally, the tagging with i of all moves in $\sigma_{(j,i)}$, where $\theta_j = \sigma_{(j,0)} \to \cdots \to \sigma_{(j,k_j)} \to \sigma'_j$
- the unique tagging with j of all moves in σ_j.

Lemma 30 (Term representation). *For any second-order IA term*:

$$[\![\Gamma \vdash M : \theta]\!] \overset{\rho_1}{\cong} [\![\Gamma \vdash M : \theta]\!]^{\mathsf{comp}}.$$

Proof (by induction on the derivation of $\Gamma \vdash M : \theta$).
Language constants. The sets of complete plays for

$$\mathbf{k} ::= n \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{skip} \mid \mathbf{diverge}$$

are finite and we can see by inspection that $[\![\mathbf{k}]\!] = [\![\mathbf{k}]\!]^{\mathsf{comp}}$ in each case.
Identifiers. Consider the set of plays of the copy-cat strategy for the projection $\pi_x : [\![\Gamma]\!] \to [\![\theta]\!]$:

$$\Sigma_{\pi_x} = \{\mathsf{s} \in \mathsf{P}_{[\![\theta]\!]_1 \multimap [\![\theta]\!]_2} \mid \text{if } \mathsf{s}' \sqsubseteq \mathsf{s}, \ \text{even}(\text{length}(\mathsf{s}')), \ \text{then } \mathsf{s}' \upharpoonright [\![\theta]\!]_1 = \mathsf{s}' \upharpoonright [\![\theta]\!]_2\}.$$

Its subset of complete plays, $\Sigma^{\mathsf{comp}}_{\pi_x}$, is a regular language isomorphic to

$$\mathscr{K}^x_\theta = \mathscr{R}[\![\theta]\!][\mathsf{m}_\mathsf{o}/\mathsf{m}_\mathsf{o} \cdot \mathsf{m}_\mathsf{o}^{\langle x \rangle}][\mathsf{m}_\mathsf{p}/\mathsf{m}_\mathsf{p}^{\langle x \rangle} \cdot \mathsf{m}_\mathsf{p}]$$

for all moves $\mathsf{m}_\mathsf{o}, \mathsf{m}_\mathsf{p} \in \mathsf{M}_\theta$, such that m_o is any opponent move and m_p any player move. It is easy to check that the isomorphism between these sets is ρ_1 as required.
Abstraction. The semantic interpretation of abstraction has the property that

$$[\![\Gamma \vdash \lambda x{:}\sigma.P : \sigma \to \theta]\!]^{\mathsf{comp}} = \Lambda([\![\Gamma, x : \sigma \vdash P : \theta]\!]^{\mathsf{comp}}) \cong [\![\Gamma, x : \sigma \vdash P : \theta]\!]^{\mathsf{comp}},$$

where $\Lambda(-)$ is an isomorphism. Therefore, using the induction hypothesis,

$$[\![\Gamma, x : \sigma \vdash P : \theta]\!]^{\mathsf{comp}} \overset{\rho_1}{\cong} [\![\Gamma, x : \sigma \vdash P : \theta]\!].$$

Also,

$$[\![\Gamma, x : \sigma \vdash P : \theta]\!] \cong [\![\Gamma, x : \sigma \vdash P : \theta]\!] \upharpoonright [\alpha^{\langle x \rangle}/\alpha^{\langle 1 \rangle}] = [\![\Gamma \vdash \lambda x{:}\sigma.P : \sigma \to \theta]\!]$$

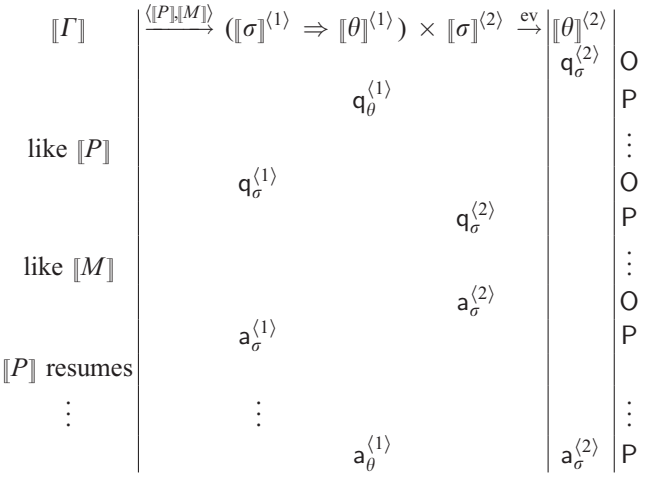


Fig. 7. Plays of function application.

for all $\alpha \in [\![\sigma]\!]$; this is because the increment $(-)\uparrow$ is an isomorphism and the substitution $[\alpha^{\langle x\rangle}/\alpha^{\langle 1\rangle}]$ also an isomorphism. The composition of these two isomorphisms can be easily seen to be ρ_1^{-1}.

Application. We need to prove that for any $\Gamma \vdash P : \sigma \to \theta, M : \sigma$,

$$[\![PM : \theta]\!]^{\mathsf{comp}} \overset{\rho_1}{\cong} ([\![\Gamma \vdash P : \sigma \to \theta]\!][\kappa])\downarrow,$$

where $\kappa(\mathsf{q}^{\langle 1\rangle} \cdot \mathsf{a}^{\langle 1\rangle}) = \{w \mid \mathsf{q} \cdot w \cdot \mathsf{a} \in [\![M : \sigma]\!]\}$, for any $\mathsf{q} \in \mathsf{Q}[\![\sigma]\!]$, $\mathsf{a} \in \mathsf{A}_{\mathsf{Q}}[\![\sigma]\!]$. The game semantic interpretation of application of $P : \sigma \to \theta$ to $M : \sigma$ is

$$[\![\Gamma]\!] \xrightarrow{\langle [\![P]\!],[\![M]\!]\rangle} [\![\sigma]\!] \Rightarrow [\![\theta]\!] \times [\![\sigma]\!] \xrightarrow{\mathrm{ev}} [\![\theta]\!],$$

where *ev* is the *evaluation* strategy. In general, this strategy does not have a regular set of complete plays, so the proof will be made directly at the level of the game semantics, by analysing all the possible plays (Fig. 7).

The opening question ($\mathsf{q}_\theta^{\langle 2\rangle}$) always occurs in $[\![\theta]\!]^{\langle 2\rangle}$, then is copied to $[\![\theta]\!]^{\langle 1\rangle}$ by *ev* (as $\mathsf{q}_\theta^{\langle 1\rangle}$). Subsequently, the strategy for $[\![P]\!]$ takes control of the play and it holds control of the play until a move ($\mathsf{q}_\sigma^{\langle 1\rangle}$) occurs in $[\![\sigma]\!]^{\langle 1\rangle}$. This move transfers control back to *ev*, which copies it to $[\![\sigma]\!]^{\langle 2\rangle}$ (as $\mathsf{q}_\sigma^{\langle 1\rangle}$). Subsequent play is then controlled by $[\![M]\!]$.

This is where the restriction to first order for P (ground type for M) ensures that application is correctly represented by regular-language homomorphism (substitution). Two observations are essential:

(1) a next move to $[\![\theta]\!]^{\langle 1\rangle}$ is not possible because the play is governed by $[\![M]\!]$, which cannot use that type component;
(2) since σ is a ground type, $[\![M]\!]$ must complete its play after moves in $[\![\Gamma]\!]$ only. A higher-order strategy would be able to ask a question in $[\![\sigma]\!]^{\langle 2\rangle}$, which *ev*

would copy back to $[\![\sigma]\!]^{\langle 1\rangle}$, and give control back to $[\![P]\!]$. This would cause a "nesting" of plays, and hence non-regularity, rather than the simple interleaving of first-order application.

Once $[\![M]\!]$ completes, the answer $\mathsf{a}_\sigma^{\langle 2\rangle}$ is copied by *ev* from $[\![\sigma]\!]^{\langle 2\rangle}$ to $[\![\sigma]\!]^{\langle 1\rangle}$ and control switches back to $[\![P]\!]$, which resumes play from where it left off.

Finally, once $[\![P]\!]$ produces an answer $\mathsf{a}_\theta^{\langle 1\rangle}$, it is relayed by *ev* to $[\![\theta]\!]^{\langle 2\rangle}$, closing the entire play.

We can see how the moves from the two σ components function as switches between the strategies of $[\![P]\!]$ and $[\![M]\!]$, inserting complete plays of $[\![M]\!]$ bracketed by $\mathsf{q}_\sigma^{\langle 2\rangle}$ and $\mathsf{a}_\sigma^{\langle 2\rangle}$ in the plays of $[\![P]\!]$, whenever moves $\mathsf{q}_\sigma^{\langle 1\rangle} \cdot \mathsf{a}_\sigma^{\langle 1\rangle}$ occur.

Finally, all the moves from components $[\![\sigma]\!]^{\langle 1\rangle} \Rightarrow [\![\theta]\!]^{\langle 1\rangle} \times [\![\sigma]\!]^{\langle 2\rangle}$ are hidden, resulting in the same regular language as the one defined by substitution.

From the induction hypothesis, $[\![P]\!]^{\mathsf{comp}} \overset{\rho_1}{\cong} [\![P]\!]$, $[\![M]\!]^{\mathsf{comp}} \overset{\rho_1}{\cong} [\![M]\!]$. Decrement $(-)\downarrow$ is also an isomorphism, necessary in order to re-associate moves with the proper components after the elimination of σ.

Term-forming expressions. For sequential composition of commands, we need to prove that $[\![\Gamma \vdash C; C']\!] \overset{\rho_1}{\cong} [\![\Gamma \vdash \mathtt{seq}\, CC']\!]^{\mathsf{comp}}$, where

$$\mathtt{seq} \overset{\text{def}}{=} \lambda c\colon \mathbf{comm}.\lambda c'\colon \mathbf{comm}.c; c'.$$

The regular-language interpretation of `seq` is

$$[\![\mathtt{seq}\colon \mathbf{comm} \to \mathbf{comm} \to \mathbf{comm}]\!] = run \cdot run^{\langle 1\rangle} \cdot done^{\langle 1\rangle} \cdot run^{\langle 2\rangle} \cdot done^{\langle 2\rangle} \cdot done.$$

It is easy to see that $[\![\mathtt{seq}\, CC']\!] = [\![C; C']\!]$ and that $[\![\mathtt{seq}]\!] \overset{\rho_1}{\cong} [\![\mathtt{seq}]\!]^{\mathsf{comp}}$. Applying the induction hypothesis, $[\![C]\!] \overset{\rho_1}{\cong} [\![C]\!]^{\mathsf{comp}}$ and $[\![C']\!] \overset{\rho_1}{\cong} [\![C']\!]^{\mathsf{comp}}$, together with the correctness of our semantics of application completes the proof.

The other term-forming expressions have similar proofs.

Iteration. The game-semantics model of iteration is in terms of recursion, which our language fragment lacks. However, it is easy to show that

$$[\![\mathbf{while}\ B\ \mathbf{do}\ C]\!]^{\mathsf{comp}} = \bigcup_{i\in\mathbb{N}} [\![W_i]\!]^{\mathsf{comp}},$$

where $W = \Gamma \vdash \lambda c_0\colon \mathbf{comm}.\mathbf{if}\ B\ \mathbf{then}\ C; c_0\ \mathbf{else}\ \mathbf{skip}$, $W_0 = \mathbf{diverge}$ and $W_{i+1} = W(W_i)$ where c_0 is not free in B or C.

A straightforward induction using the correctness of application and branching shows that $[\![W_i]\!] \overset{\rho_1}{\cong} [\![W_i]\!]^{\mathsf{comp}}$.

We therefore only need to show that $[\![\mathbf{while}\ B\ \mathbf{do}\ C]\!] = \bigcup_{i\in\mathbb{N}} [\![W_i]\!]$, i.e. that

$$(\!|W_n|\!) = \sum_{i=0}^{n-1} \underbrace{(\!|B|\!)_{tt} \cdot (\!|C|\!) \cdots (\!|B|\!)_{tt} \cdot (\!|C|\!)}_{i \text{ times}} \cdot (\!|B|\!)_{ff},$$

which we do by induction on n. The base case is trivial: $(\!| W_0 |\!) = \emptyset$ by definition. For the inductive step we calculate as follows:

$$\begin{aligned}(\!| W_{n+1} |\!) &= (\!| W(W_n) |\!) \\ &= (\!| W |\!)[run^{\langle 1\rangle} \cdot done^{\langle 1\rangle}/(\!| W_n |\!)] \downarrow \\ &= ((\!| B |\!)_{tt} \cdot (\!| C |\!) \cdot run^{\langle 1\rangle} \cdot done^{\langle 1\rangle} + (\!| B |\!)_{ff})[run^{\langle 1\rangle} \cdot done^{\langle 1\rangle}/(\!| W_n |\!)] \downarrow \\ &= (\!| B |\!)_{tt} \cdot (\!| C |\!) \cdot (\!| W_n |\!) + (\!| B |\!)_{ff}\end{aligned}$$

from which the result follows.

Local variables. According to the game-semantic definition of `new`, a complete play of the strategy interpreting it is

$$\begin{array}{lll} (\mathbf{var}\tau \to \sigma) & \to \sigma & \\ & & \mathsf{q} \\ & \mathsf{q} & \\ \mathsf{s} & & \\ & \mathsf{a} & \\ & & \mathsf{a} \end{array}$$

where s is a sequence of moves in which all occurrences of *read* and *write* globally satisfy the constraints made formal by the stability regular expression $\gamma_{\mathbf{var}\,\tau}$. Using the game-semantic definition of composition and an analysis of possible moves similar to the one we did for application, it follows that $[\![\mathbf{new}_\tau(\lambda x{:}\,\mathbf{var}\,\tau.M)]\!]^{\mathsf{comp}} \overset{\rho_1}{\cong} [\![\mathbf{new}\,\tau\ x\ \mathbf{in}\ M]\!]$. □

We can now give the proof of Theorem 27.

Proof of Theorem 27. The full abstraction result for the game model of IA states that two IA terms are equivalent iff they have the same sets of complete plays (Theorem 2). By our earlier result (Proposition 1) this immediately restricts to second-order IA. The term representation lemma proves that the regular language semantics is a correct representation of the game semantics, hence it establishes the full abstraction theorem for the regular language semantics of second-order IA. □

7. Semantics of full second-order IA

We now consider the semantics of the binding construct **let**. Without this construct the language fragment cannot function as a stand-alone programming language: it contains identifiers of function types but no way to bind them to actual functions. The incorporation of **let** resolves this issue and turns our language fragment into a viable stand-alone programming language.

At ground types, the definition of *let* is redundant because it can be replaced by abstraction and application: **let** x **be** P **in** $P' = (\lambda x{:}\,\sigma.P')P$. But this redundancy does not create any technical difficulties.

Semantically, the most straightforward way to handle binding is extensionally, by adding an *environment* u as a parameter to the semantic valuation function. The environment is a function mapping the free identifiers of the term to regular languages:

Definition 31 (Binding).

$$[\![\Gamma \vdash \textbf{let } x \textbf{ be } P \textbf{ in } P' : \theta]\!]u = [\![\Gamma, x : \theta' \vdash P' : \theta]\!](u \,|\, x \mapsto [\![\Gamma \vdash P' : \theta']\!]u).$$

The interpretation of identifiers will be different in the presence of the environment.

Definition 32 (Identifiers). $[\![\Gamma \vdash x : \theta]\!]u = u(x)$.

All the other semantic definitions of the previous section stay the same, except that all functions $[\![-]\!], (\!|-|\!)$ now take the additional parameter u.

The following property is technically important:

Lemma 33 (Term substitution).

$$[\![\Gamma \vdash \textbf{let } x \textbf{ be } P \textbf{ in } P' : \theta]\!]u = [\![\Gamma \vdash P'[x/P]]\!]u.$$

Proof (by structural induction on the syntax of P').

Basis. If $P' = \mathbf{k}$ is an IA constant, then $\mathbf{k} = \mathbf{k}[x/P]$ and the property follows trivially.

If $P' = x'$ is an identifier then we have two cases:

either $(x = x')$, in which case $x[x/P] = P$ and

$$\begin{aligned}[\![\Gamma \vdash \textbf{let } x \textbf{ be } P \textbf{ in } x : \theta]\!]u &= [\![\Gamma, x : \theta \vdash x : \theta]\!](u \,|\, x \mapsto [\![\Gamma \vdash P : \theta]\!]u) \\ &= (u \,|\, x \mapsto [\![\Gamma \vdash P : \theta]\!]u)(x) \\ &= [\![\Gamma \vdash P : \theta]\!]u = [\![\Gamma \vdash x[x/P] : \theta]\!]u\end{aligned}$$

or $(x \neq x')$, in which case $x'[x/P] = x'$ and

$$\begin{aligned}[\![\Gamma \vdash \textbf{let } x \textbf{ be } P \textbf{ in } x' : \theta]\!]u &= [\![\Gamma, x' : \theta \vdash x' : \theta]\!](u \,|\, x \mapsto [\![\Gamma \vdash P : \theta']\!]u) \\ &= [\![\Gamma, x' : \theta \vdash x' : \theta]\!]u = [\![\Gamma \vdash x'[x/P] : \theta]\!]u.\end{aligned}$$

Composite terms. For non-binding terms of IA (i.e. all except *let* and abstraction) the proofs are similar. We present sequential composition in detail: $[\![\textbf{let } x \textbf{ be } P \textbf{ in } C; C']\!]u$ is equal to

$$\begin{aligned}&[\![C; C']\!](u \,|\, x \mapsto [\![P]\!]u) \\ &\quad = run \cdot (\!|\, C \,|\!)(u \,|\, x \mapsto [\![P]\!]u) \cdot (\!|\, C' \,|\!)(u \,|\, x \mapsto [\![P]\!]u) \cdot done \\ &\quad = run \cdot (\!|\, C[x/P] \,|\!)u \cdot (\!|\, C'[x/P] \,|\!)u \cdot done \quad \text{(by induction hyp.)} \\ &\quad = run \cdot (\!|\, C[x/P]; C'[x/P] \,|\!)u \cdot done\end{aligned}$$

$$= run \cdot (\!|(C; C')[x/P]|\!)u \cdot done$$
$$= [\![(C; C')[x/P]]\!]u.$$

For the two binding combinators we have two cases:
- $x \neq x'$: similar to the one above, for non-binding combinators.
- $x = x'$: then $[\![\textbf{let } x \textbf{ be } P \textbf{ in let } x \textbf{ be } P' \textbf{ in } P'']\!]u = [\![\textbf{let } x \textbf{ be } P' \textbf{ in } P'']\!]u$. □

It is intuitively clear that the definition of **let** is orthogonal to the purely regular-language semantics of the previous section. This intuition is formalized by the following property:

Lemma 34 (Reduction). *For any term $\Gamma \vdash P : \theta$ of IA there exists a **let**-free term $\Gamma \vdash P_0 : \theta$ such that $\Gamma \vdash P \equiv_\theta P_0$ and $[\![\Gamma \vdash P : \theta]\!]u_\Gamma = [\![\Gamma \vdash P_0 : \theta]\!]$, where u_Γ is an environment mapping all identifiers of Γ to copy-cat regular expressions: $dom(u_\Gamma) = dom(\Gamma)$, and $u_\Gamma(x) = \mathscr{K}^x_{\Gamma(x)}$.*

The overloaded notations $[\![-]\!]u$ and $[\![-]\!]$ should not create confusion: the former is the environment-based semantics of this section, the latter is the purely regular-language semantics of the previous section. First we prove the following ancillary result:

Proposition 35. *If $\Gamma \vdash P : \theta$ of IA is **let**-free, then $[\![\Gamma \vdash P : \theta]\!]u_\Gamma = [\![\Gamma \vdash P : \theta]\!]$.*

Proof. The proof is by an easy induction on the syntax of P. The only interesting case is the base case when P is an identifier x. Then by definition of the semantics,

$$[\![x]\!] = \mathscr{K}^x_\theta = u_\Gamma(x) = [\![x]\!]u_\Gamma. \qquad \square$$

The main proof of the Reduction Lemma is by induction on the number of occurrences of **let** in P:

Proof.
Basis. $(n = 0)$ By previous proposition.
Inductive step. We use an inner induction on the structure of P. Base cases are trivial since they do not contain **let**. For composite terms other than **let**-terms, the inductive hypothesis together with compositionality of both direct and environment-based semantics give the result directly.

Finally, for terms of form **let** x **be** P' **in** P'' we argue as follows. By the inductive hypothesis there is some **let**-free term $P'_0 \equiv P'$ such that $[\![P']\!]u_\Gamma = [\![P'_0]\!] = [\![P'_0]\!]u_\Gamma$, using the previous proposition. The term reduces as follows:

$$\begin{aligned}
[\![\textbf{let } x \textbf{ be } P' \textbf{ in } P'']\!]u_\Gamma &= [\![P'']\!](u_\Gamma \,|\, x \mapsto [\![P']\!]u_\Gamma) \\
&= [\![P'']\!](u_\Gamma \,|\, x \mapsto [\![P'_0]\!]u_\Gamma) \\
&= [\![\textbf{let } x \textbf{ be } P'_0 \textbf{ in } P'']\!]u_\Gamma \\
&= [\![P''[x/P'_0]]\!]u_\Gamma
\end{aligned}$$

by the substitution lemma. But $P''[x/P'_0]$ contains fewer occurrences of **let** than does **let** x **be** P' **in** P'' and is clearly operationally equivalent to it. We can apply the induction hypothesis to conclude. □

We can now state the two principal properties of the regular-language semantics of second-order IA.

Theorem 36 (Full abstraction).

$$\Gamma \vdash P_1 \equiv_\theta P_2 \quad \textit{if and only if} \quad [\![\Gamma \vdash P_1 : \theta]\!]u_\Gamma = [\![\Gamma \vdash P_2 : \theta]\!]u_\Gamma.$$

Proof. Given terms P_1, P_2, by the reduction Lemma 34, we can find **let**-free terms P'_1, P'_2, such that $P_i \equiv P'_i$ and $[\![P_i]\!]u_\Gamma = [\![P'_i]\!]u_\Gamma = [\![P'_i]\!]$. Therefore, by the previous full-abstraction result (Theorem 27):

$$P_1 \equiv P_2 \;\Leftrightarrow\; P'_1 \equiv P'_2 \;\Leftrightarrow\; [\![P'_1]\!] = [\![P'_2]\!] \;\Leftrightarrow\; [\![P_1]\!]u_\Gamma = [\![P_2]\!]u_\Gamma. \qquad \square$$

An immediate corollary of the full abstraction theorem is

Corollary 37 (Decidability). Equivalence of second-order finitary IA terms is decidable.

Proof. The fully abstract regular-language semantics interprets terms as regular languages, for which equivalence is decidable. □

This result is related to much earlier results of Jones regarding decidability properties of programs written in a similar programming language [14]. Our decidability result is more general because it concerns program *fragments* rather than *full* programs.

The language described here supports some straightforward extensions such as arrays or low-level data pointers (*à la* C) which can be introduced as syntactic sugar. A simple *semantic* extension, which, for this second-order fragment is orthogonal to the rest of the semantic model, is *bounded non-determinism* [12]. The only addition to the language is a non-deterministic expression:

$$[\![\mathbf{random}\ \tau : \mathbf{exp}\,\tau]\!]u = \sum_{\alpha \in \mathscr{A}[\![\tau]\!]} q \cdot \alpha.$$

A more substantial modification of the language is the use of call-by-value instead of call-by-name. The regular-language semantics of this language is presented in [6]. Other extensions, such as control or parallelism will require a much more substantial revision of the semantic framework.

8. Related and further work

The research presented in this article, originally presented in [8], has since been expanded in two directions.

The first author has proved a similar decidability result for a call-by-value language, showing that it has a regular-language model [6]. Ong has shown that observational equivalence of the third-order fragment of IA without iteration is decidable [25] by showing that its game model can be represented using deterministic context-free languages. Finally, Murawski has shown that higher-order fragments of imperative procedural languages, call-by-name or call-by-value, are not decidable [17].

Another direction of research is the application of the regular-language semantics to program verification, and it was pursued by the first author in his doctoral thesis [7]. A model-checking tool, used to generate the models in Section 5.6, based on this work is currently under development.

Acknowledgements

The authors are grateful to Robert Tennent and Samson Abramsky for suggestions, encouragement and advice. We thank Peter O'Hearn and Pasquale Malacaria for several stimulating discussions on the subject matter. Steve Brookes's careful reading of a previous version of this material [7] and the insightful comments of the anonymous referees for the conference paper that preceded this article [8] have greatly contributed to improving the presentation.

References

[1] S. Abramsky, G. McCusker, Linearity, sharing and state: a fully abstract game semantics for idealized Algol with active expressions (extended abstract), in: Proc. of 1996 Workshop on Linear Logic, Vol. 3, Electronic notes in Theoretical Computer Science, Elsevier, Amsterdam, 1996 (also as Chap. 20 of [22]).

[2] S. Abramsky, G. McCusker, Game semantics, Lecture Notes, Marktoberdorf summer school, 1997 (available from http://web.comlab.ox.ac.uk/oucl/work/samson.abramsky/mdorf97.ps.gz).

[3] S. Brookes, Full abstraction for a shared variable parallel language, in: Proc. 8th Annual IEEE Symp. on Logic in Computer Science, IEEE Computer Society Press, Los Alamitos, CA, Montreal, Canada, 1993, pp. 98–109 (published also as Chap. 21 of [22]).

[4] N. Bruijn, Lambda calculus notation with namefree formulas involving symbols that represent reference transforming mappings, Indag. Math. 40 (3) (1979) 348–356.

[5] D.R. Ghica, A regular-language model for Hoare-style correctness statements, in: Proc. of the Verification and Computational Logic 2001 Workshop, Florence, Italy, 2001.

[6] D.R. Ghica, Regular language semantics for a call-by-value programming language, in: Proc. of the 17th Annual Conf. on Mathematical Foundations of Programming Semantics, Electronic Notes in Theoretical Computer Science, Elsevier, Aarhus, Denmark, 2001, pp. 85–98.

[7] D.R. Ghica, A games-based foundation for compositional software model checking, Ph.D. Thesis, Queen's University School of Computing, Kingston, Ontario, Canada; also available as Oxford University Computing Laboratory Research Report RR-02-13, November 2002.

[8] D.R. Ghica, G. McCusker, Reasoning about Idealized ALGOL using regular languages, in: Proc. of 27th Internat. Coll. on Automata, Languages and Programming ICALP 2000, Lecture Notes in Computer Science, Vol. 1853, Springer, Berlin, 2000, pp. 103–116.

[9] C. Hankin, P. Malacaria, Generalised flowcharts and games, Lecture Notes in Computer Science, Vol. 1443, Springer, Berlin, 1998.

[10] C. Hankin, P. Malacaria, A new approach to control flow analysis, Lecture Notes in Computer Science, Vol. 1383, Springer, Berlin, 1998.

[11] D. Harel, First-order dynamic logic, Lecture Notes in Computer Science, Vol. 68, Springer, New York, NY, USA, 1979 (rev. version of the author's thesis, MIT, 1978).
[12] R. Harmer, G. McCusker, A fully abstract game semantics for finite nondeterminism, in: 14th Symp. on Logic in Computer Science (LICS'99), IEEE, Washington, Brussels, Tokyo, 1999, pp. 422–430.
[13] M. Jantzen, Extending regular expressions with iterated shuffle, Theoret. Comput. Sci. 38 (2–3) (1985) 223–247.
[14] N.D. Jones, S.S. Muchnick, Even simple programs are hard to analyze, J. Assoc. Comput. Mach. 24 (2) (1977) 338–350.
[15] G. McCusker, Games and Full Abstraction for a Functional Metalanguage with Recursive Types, Distinguished Dissertations, Springer, Berlin, 1998.
[16] A.R. Meyer, K. Sieber, Towards fully abstract semantics for local variables: preliminary report, in: Conf. Record of the 15th Annual ACM Symp. on Principles of Programming Languages, ACM, New York, San Diego, CA, 1988, pp. 191–203 (reprinted as Chap. 7 of [22]).
[17] A. Murawski, On program equivalence in languages with ground-type references, in: Proc. of IEEE Symp. on Logic in Computer Science, 2003, to appear.
[18] P.W. O'Hearn, A.J. Power, M. Takeyama, R.D. Tennent, Syntactic control of interference revisited, Theoret. Comput. Sci. 228 (1999) 175–210 (preliminary version reprinted as Chap. 18 of [22]).
[19] P.W. O'Hearn, U.S. Reddy, Objects, interference and the Yoneda embedding, in: S. Brookes, M. Main, A. Melton, M. Mislove (Eds.), Mathematical Foundations of Programming Semantics, 11th Annual Conference, Electronic Notes in Theoretical Computer Science, Vol. 1, Elsevier Science, Tulane University, New Orleans, Louisiana, 1995.
[20] P.W. O'Hearn, J.C. Reynolds, From Algol to polymorphic linear lambda-calculus, J. Assoc. Comput. Mach. 47 (1) (2000) 167–223.
[21] P.W. O'Hearn, R.D. Tennent, Relational parametricity and local variables, in: Conf. Record of the 20th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, ACM, New York, Charleston, South Carolina, 1993, pp. 171–184 (a version also published as Chap. 16 of [22]).
[22] P.W. O'Hearn, R.D. Tennent (Eds.), ALGOL-like Languages, Progress in Theoretical Computer Science, Birkhäuser, Boston, 1997, two volumes.
[23] F.J. Oles, A category-theoretic approach to the semantics of programming languages, Ph.D. Thesis, Syracuse University, Syracuse, NY, 1982.
[24] F.J. Oles, Functor categories and store shapes, in: P.W. O'Hearn, R.D. Tennent (Eds.), ALGOL-like Languages, Progress in Theoretical Computer Science, Birkhäuser, Boston, 1997, Chap. 11, pp. 3–12.
[25] C.-H.L. Ong, Observational equivalence of third-order idealized Algol is decidable, in: Proc. of IEEE Symp. on Logic in Comp. Science, 2002, pp. 245–256.
[26] A.M. Pitts, Reasoning about local variables with operationally-based logical relations, in: 11th Annual Symp. on Logic in Computer Science, IEEE Computer Society Press, Washington, 1996, pp. 152–163 (a version also published as Chap. 17 of [22]).
[27] U.S. Reddy, Global state considered unnecessary: introduction to object-based semantics, LISP Symbolic Comput. 9 (1) (1996) 7–76 (published also as Chap. 19 of [22]).
[28] J.C. Reynolds, Syntactic control of interference, in: Conf. Record of the 5th Annual ACM Symp. on Principles of Programming Languages, ACM, New York, Tucson, Arizona, 1978, pp. 39–46.
[29] J.C. Reynolds, The essence of ALGOL, in: J.W. de Bakker, J.C. van Vliet (Eds.), Algorithmic Languages, Proc. Internat. Symp. on Algorithmic Languages, North-Holland, Amsterdam, 1981, pp. 345–372 (reprinted as Chap. 3 of [22]).
[30] J.C. Reynolds, The Craft of Programming, Prentice-Hall International, London, 1981.
[31] J.C. Reynolds, Theories of Programming Languages, Cambridge University Press, Cambridge, MA, 1998.
[32] D.A. Schmidt, On the need for a popular formal semantics, ACM SIGPLAN Notices 32 (1) (1997) 115–116.

[33] D.S. Scott, C. Strachey, Toward a mathematical semantics for computer languages, in: J. Fox (Ed.), Proc. of the Symp. on Computers and Automata, Microwave Research Institute Symposia Series, Vol. 21, Polytechnic Institute of Brooklyn Press, New York, 1971, pp. 19–46; also Technical Monograph PRG-6, Oxford University Computing Laboratory, Programming Research Group, Oxford.

[34] K. Sieber, Full abstraction for the second order subset of an ALGOL-like language, in: Mathematical Foundations of Computer Science, Vol. 841, Lecture Notes in Computer Science, Springer, Berlin, Kŏsice, Slovakia, 1994, pp. 608–617 (a version also published as Chap. 15 of [22]).