Reasoning about Idealized ALGOL Using Regular Languages

Dan R. Ghica[1,*] and Guy McCusker[2]

[1] Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, K7L 3N6; Fax: +1 (613) 533-6513; ghica@cs.queensu.ca
[2] School of Cognitive and Computing Sciences, University of Sussex at Brighton, Falmer, Brighton, UK, BN1 9QH; Fax: +44 (1273) 671320; guym@cogs.susx.ac.uk

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 method of reasoning inherits its desirable theoretical properties. The method is mathematically elementary and formal, which makes it uniquely suitable for automation. We show that reasoning can be carried out using only a meta-language of extended regular expressions, a language for which equivalence is formally 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 procedural 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 [14]. 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 [10].

A common theme in the literature on semantics of IA, beginning with [5], 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 [11], relational parametricity was employed to capture representation-independence properties [9], and linear logic to explain irreversibility [8]. Recently, many of these ideas have been successfully incorporated in an operationally-based account of IA by Pitts [12].

* This author acknowledges the support of a PGSB grant from the Natural Sciences and Engineering Research Council of Canada. This paper was written while visiting University of Edinburgh, Laboratory for Foundations of Computer Science.

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 correct equivalences between programs, but unfortunately the model as originally presented is complicated, and calculating and reasoning within the model is difficult.

In this paper, we show that if one restricts 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 therefore able to validate them in a largely calculational, algebraic style, using our semantics. 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" [13] and Brookes's trace semantics for shared-variable concurrent ALGOL [2]. Identifiers are not interpreted using an environment, variables are not interpreted using a notion of store and functions in the language are not interpreted using a mathematical notion of function. Instead, we are primarily concerned with behaviour, with all the possible actions that can be associated with every such 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" [16]; 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. 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 [3, 4]. They used such models to derive static analysis algorithms which can be described without reference to games.

2 The IA Fragment

The principles of the programming language IA were laid down by John Reynolds in an influential paper [15]. 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 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 analyzed extensively [18, 1, 8]. 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 of the language (*i.e.* types of data assignable to variables) are a finite subset of the integers, and booleans:

$$\tau ::= \mathbf{int} \mid \mathbf{bool}$$

The phrase types of the language are those of commands, variables and expressions, plus function types.

$$\sigma ::= \mathbf{comm} \mid \mathbf{var}[\tau] \mid \mathbf{exp}[\tau] , \qquad \theta ::= \sigma \mid \sigma \rightarrow \theta$$

Note that we include only first-order function types here. We will consider only terms of the form

$$\iota_1 : \theta_1, \ldots, \iota_k : \theta_k \vdash M : \sigma$$

that is, terms of ground type with free variables of arbitrary first-order type. For the sake of simplicity in this paper, we also assume that M is β-normal, so that it contains no λ-abstractions. Function application is restricted to free identifiers ι. This last restriction can easily be removed, but at the expense of undue notational overhead in the semantics.

The terms of the language are as follows. In type **comm** there are basic commands **skip**, to do nothing, and Ω to diverge; in type **exp**[**int**] the finitary fragment contains constants n belonging to a finite subset $\mathcal{N}$ of the set of integers; and in type **exp**[**bool**] there are the constants **true** and **false**. There are term formers for assignment to variables, $V := E$, dereferencing variables, $!V$, sequential composition of commands $C; C'$, and sequential composition of a command with an expression to yield a possibly side-effecting expression $C; E$. We have a conditional operation **if** B **then** C **else** C', a while-loop **while** B **do** C, application of first-order identifiers to arguments $\iota M_1 \ldots M_k$, and the local-variable declaration **new**[τ] ι **in** C. Here, the free variable $\iota : \mathbf{var}[\tau]$ of C becomes bound. Finally, we assume the usual range of binary operations on integer and boolean expressions.

3 Game Semantics of Idealized ALGOL

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

$$\iota : \mathbf{exp}[\mathbf{int}] \rightarrow \mathbf{comm} \vdash M : \mathbf{comm} ,$$

Player will represent the program M; *Opponent* represents the context, in this case the non-local procedure ι. 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** might be

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

In more interesting games, such as the one used to interpret programs like

$$\iota : \mathbf{exp}[\mathbf{int}] \to \mathbf{comm} \vdash \iota(0) : \mathbf{comm} \ ,$$

there are more moves. Corresponding to the result type **comm**, there are the moves *run* and *done*. The program needs to run the procedure ι, so there are also moves run_ι and $done_\iota$ to represent that; here the run_ι move is a move for P, and $done_\iota$ is a move for O. Finally, the procedure ι may need to evaluate its argument. For this purpose, O has a move q^1_ι, meaning "what is the value of the first argument to ι?", to which P may respond with an integer n, tagged as n^1_ι for the sake of identification.

Here is a sample interaction in the interpretation of the above term.

O: *run* (start executing)
P: run_ι (execute ι)
O: q^1_ι (what is the first argument to ι?)
P: 0^1_ι (the argument is 0)
O: $done_\iota$ (ι terminates)
P: *done* (whole command terminates).

In the above interaction, at the third move, O was not compelled to ask for the argument to ι: if O represented a non-strict procedure, the move $done_\iota$ would be played immediately. Similarly, at the fifth move, O could repeat the question q_ι to represent a procedure which calls its argument more than once.

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, that is, 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 Ω 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 $\iota(0)$, for example, is capable of supplying the argument 0 to ι as often as O asks for it.

Interpretation of Variables. The type $\mathbf{var}[\tau]$ is represented as a game in the following way. For each element x of τ there is an initial move $write(x)$, representing an assignment. There is one possible response to this move, *ok*, which signals successful completion of the assignment. For dereferencing, there is an initial move *read*, to which P may respond with any element of τ.

Here is an interaction in the strategy for

$$v : \mathbf{var}[\mathbf{int}] \vdash v :=!v + 1.$$

O: *run*
P: $read_v$ (get the value from v)
O: 3 (O supplies the value 3)
P: $write(4)_v$ (write 4 into v)
O: ok_v (the assignment is complete)
P: *done* (the whole command is complete)

In these interactions, O is *not* constrained to play a *good variable* in v, *i.e.* to exhibit the expected causal dependency between reads and writes. For example, in the game for terms of the form

$$c : \mathbf{comm}, v : \mathbf{var}[\mathbf{int}] \vdash M : \mathbf{comm} \ ,$$

we find interactions such as

$$run \cdot read_v \cdot 3_v \cdot write(4)_v \cdot ok_v \cdot run_c \cdot done_c \cdot read_v \cdot 7_v \cdots$$

Here O has not played a good variable in v, but this freedom is necessary. Our semantics must take care of the case in which ι is bound to a procedure which also uses v, for example, the procedure $v := 7$.

There is one situation in which this kind of interference cannot 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 the interaction with v is not an observable part of the programs behaviour. Therefore, the games interpretation of **new** 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, and deleting all the moves pertaining to v, to hide v from the outside.

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, $run{\cdot}run_\iota$ is not complete but $run{\cdot}run_\iota{\cdot}done_\iota{\cdot}done$ is. Then we have the following theorem:

Theorem 1 (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.*

Note. In the above account, a very simple notion of game has been used. In fact, games models require a great deal more machinery, including the notions of *justification pointer* and *questions and answers*, 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; it only comes into play at third-order and above.

4 Regular Language Game Semantics

We will now give a simple presentation of the game semantics of our fragment of IA. The key idea is that the set of complete plays in a strategy forms a regular language, which leads to a compact notation for defining and manipulating these infinite sets of sequences. We define a metalanguage based on regular expressions, extended with two handy operations: intersection and hiding. Of course, these extensions do not change the regular nature of the languages being defined.

Definition 1. *The set $\mathcal{R}_\mathcal{A}$ of extended regular expressions over a finite alphabet $\mathcal{A}$ is defined inductively as the smallest set for which:*

Constants: $\bot, \epsilon \in \mathcal{R}_\mathcal{A}$; *if* $a \in \mathcal{A}$, *then* $a \in \mathcal{R}_\mathcal{A}$;
Iteration: *if* $R \in \mathcal{R}_\mathcal{A}$, $R^* \in \mathcal{R}_\mathcal{A}$;
Operators: *if* $R, S \in \mathcal{R}_\mathcal{A}$, *then* $R \cdot S, R + S, R \cap S \in \mathcal{R}_\mathcal{A}$;
Hiding: *if* $R \in \mathcal{R}_\mathcal{A}$, $\mathcal{A}' \subseteq \mathcal{A}$, *then* $R \mid_{\mathcal{A}'} \in \mathcal{R}_\mathcal{A}$;

The constant $\bot$ denotes the empty language, while ϵ is the language consisting only of the empty string. The constant a is the language consisting of the singleton sequence a. Hiding represents the operation of restricting a language to a subset $\mathcal{A} \setminus \mathcal{A}'$ of the original alphabet $\mathcal{A}$: the language $\mathcal{L}(R \mid_{\mathcal{A}'})$ is the set of sequences in $\mathcal{L}(R)$, with all elements of $\mathcal{A}'$ deleted. The other operations (iteration, concatenation, union, intersection) are defined as usual.

Proposition 1. *Every extended regular expression denotes a regular language.*

We now give a regular language representation of the game semantics for IA. An alphabet is associated with every type in IA. They represent a semantic "domain" over which regular languages will be constructed, using extended regular expressions:

$$\begin{aligned}
&\mathcal{A}[\![\mathbf{int}]\!] = \mathcal{N}, \quad \mathcal{A}[\![\mathbf{bool}]\!] = \{true, false\}\\
&\mathcal{A}[\![\mathbf{comm}]\!] = \{run, done\},\\
&\mathcal{A}[\![\mathbf{exp}[\tau]]\!] = \{q, v \mid v \in \mathcal{A}[\![\tau]\!]\},\\
&\mathcal{A}[\![\mathbf{var}[\tau]]\!] = \{read, v, write(v), ok \mid v \in \mathcal{A}[\![\tau]\!]\},\\
&\mathcal{A}[\![\sigma_1 \to \sigma_2 \to \ldots \to \sigma_k \to \sigma]\!] = \{a^i \mid a \in \mathcal{A}[\![\sigma_i]\!], 1 \le i \le k\} \cup \mathcal{A}[\![\sigma]\!].
\end{aligned}$$

By a^k we mean a lexical operation: the creation of a new symbol by tagging the symbol a with the numeral k.

For a term of the form

$$\iota_1 : \theta_1, \iota_2 : \theta_2, \ldots \iota_k : \theta_k \vdash M : \sigma$$

we define the *context alphabet* to be the set

$$\bigcup_{1 \le j \le k} \{a_{\iota_j} \mid a \in \mathcal{A}[\![\theta_j]\!]\}$$

that is, the union of the $\mathcal{A}[\![\theta_j]\!]$ alphabets, every symbol tagged with the corresponding identifier.

The semantics of a term M as above is then a regular language of a certain form, defined as follows.

- If $\sigma = \mathbf{comm}$, $[\![M]\!] = run \cdot R_M \cdot done$.
- If $\sigma = \mathbf{exp}[\tau]$, $[\![M]\!] = \sum_{v \in \mathcal{A}[\![\tau]\!]} q \cdot R_M^v \cdot v$
- If $\sigma = \mathbf{var}[\tau]$,

$$[\![M]\!] = \sum_{v \in \mathcal{A}[\![\tau]\!]} (read \cdot R_M^v \cdot v) + \sum_{v \in \mathcal{A}[\![\tau]\!]} (write(v) \cdot S_M^v \cdot ok)$$

where R_M, R_M^v and S_M^v are regular languages over the context alphabet of the term M. The idea is that, for M of type **comm**, for example, the regular language R_M is the set of interactions with the environment that need to take place for M to terminate. Similarly, R_M^3 is the set of interactions that an expression M must have with the environment to return a value of 3, and so on. For M of type $\mathbf{var}[\tau]$, R_M^v denotes the interactions required for a value v to be read from M, and S_M^v denotes the interactions needed to write v into M.

These regular languages, denoted by R_M, R_M^v, S_M^v, form the substance of our interpretation of the language; the moves that bracket them, such as *run*, *done* for commands, are merely delimiters to indicate that a complete play has occurred. The definitions needed to interpret most of our language are given in Table 1.

$$R_{\mathbf{skip}} = \epsilon \qquad R_{\Omega} = \bot \qquad R_v^v = \epsilon \qquad R_v^{v'} = \bot \quad (v \neq v')$$

$$R_{\iota:\mathbf{comm}} = run_\iota \cdot done_\iota \qquad R^v_{\iota:\mathbf{exp}[\tau]} = q_\iota \cdot v_\iota$$

$$R^v_{\iota:\mathbf{var}[\tau]} = read_\iota \cdot v_\iota \qquad S^v_{\iota:\mathbf{var}[\tau]} = write(v)_\iota \cdot ok_\iota$$

$$R_{\mathbf{while}\ B\ \mathbf{do}\ C} = (R_B^{true} \cdot R_M)^* \cdot R_B^{false} \qquad R^n_{E_1+E_2} = \sum_{n_1+n_2=n} R^{n_1}_{E_1} \cdot R^{n_2}_{E_2}$$

$$R^{true}_{E_1=E_2} = \sum_{n \in \mathcal{N}} R^n_{E_1} \cdot R^n_{E_2} \qquad R^{false}_{E_1=E_2} = \sum_{n1 \neq n2} R^{n_1}_{E_1} \cdot R^{n_2}_{E_2}$$

$$R_{\mathbf{if}\ B\ \mathbf{then}\ C\ \mathbf{else}\ C'} = R_B^{true} \cdot R_C + R_B^{false} \cdot R_{C'} \qquad R_{C;C'} = R_C \cdot R_{C'}$$

$$R^v_{!V} = R^v_V \qquad R_{V:=M} = \sum_v R^v_M \cdot S^v_V$$

Table 1. Some semantic valuations

For instance, a trace of $V := E$ consists of *run* and *done* surrounding the effects of the assignment: first R_E^v which is the regular language denoting the interaction which leads the expression E to return value v, and then S_V^v which is the regular language denoting the interaction required to write value v into variable V.

A trace of a while-loop has the form: some number of repetitions of a trace of the guard which produces *true* followed by a complete trace of the loop body, then, finally, a single trace of the guard producing *false*. Using our semantics, we can easily demonstrate the validity of a typical while-loop equivalence:

$$\begin{aligned}
&[\![\textbf{while true do } C]\!] \\
&\quad = run \cdot \left(R^{true}_{\textbf{true}} \cdot R_C\right)^* \cdot R^{false}_{\textbf{true}} \cdot done \\
&\quad = run \cdot (\epsilon \cdot R_C)^* \cdot \bot \cdot done \\
&\quad = \bot = [\![\Omega]\!].
\end{aligned}$$

The semantics of a free identifier ι consist simply of querying the identifier. There is no need to look up the identifier in an environment, because the tagging of the trace with the name of the identifier ensures the proper correspondence between each identifier and its effects. Therefore, a notion of environment is not needed here at all.

The semantics of application and of local variables have been omitted from Table 1 because they deserve additional explanation.

Application. Let ι be a free variable of type $\sigma_1 \to \sigma_2 \to \ldots \to \sigma_k \to \textbf{comm}$, and $M_1, \ldots, M_k$ be terms of type $\sigma_1, \ldots, \sigma_k$. The interpretation of the application $\iota M_1 \ldots M_k$ depends on the moves available, which depends on the types $\sigma_1, \ldots, \sigma_k$. In the simplest case, when every σ_j is the type **comm**, we define

$$R_{\iota M_1 \ldots M_k} = run_\iota \cdot \left(\sum_{j=1}^{k} run_\iota^j \cdot R_{M_j} \cdot done_\iota^j \right)^* \cdot done_\iota .$$

To illustrate a more complex case, we give the definition of the interpretation of ιM where ι has type $\textbf{var}[\textbf{int}] \to \textbf{exp}[\textbf{int}]$.

$$R^v_{\iota M} = q_\iota \cdot \left(\sum_n read_\iota^1 \cdot R^n_M \cdot n_\iota^1 + \sum_n write(n)_\iota^1 \cdot S^n_M \cdot ok_\iota^1 \right)^* \cdot v_\iota .$$

The large sums in this expression show that the environment chooses how to read and write from the argument to ι, and that the term M determines what behaviour results from such reading and writing.

In general, for a variable $\iota : \sigma_1 \to \sigma_2 \to \ldots \sigma_k \to \textbf{comm}$:

$$R_{\iota M_1 \ldots M_k} = run_\iota \cdot \left(\sum_{j=1}^{k} \rho_\iota^j [\![M_j]\!] \right)^* \cdot done_\iota$$

where ρ_ι^j is a relabeling operation that tags the initial and final moves of the arguments M_j, the bracketing indicating a complete play, with the identifier which is calling them and the position in which they are used:

$$\rho_\iota^j(R) = R[w_\iota^j / w], \text{ for } w \in \{run, done, q, v, read, write(v), ok \mid v \in \mathcal{A}[\![\tau]\!]\}.$$

Local variables. For the semantics of a local variable block, as in the original game semantics, there are two things to do: restrict O's behaviour to that of a good variable, and hide the interaction with the local variable.

The regular language γ_ι^τ stipulates that the moves corresponding to ι have good-variable behaviour. First, let $\mathcal{A}[\![\tau]\!]_\iota$ be that part of the alphabet which concerns the variable $\iota : \mathbf{var}[\tau]$, that is,

$$\mathcal{A}[\![\tau]\!]_\iota = \{read_\iota, v_\iota, write(v)_\iota, ok_\iota \mid v \in \mathcal{A}[\![\tau]\!]\}.$$

Let $B_\iota = (\sum_{x \notin \mathcal{A}[\![\tau]\!]_\iota} x)^*$ be the regular language containing all strings which do not contain any elements of $\mathcal{A}[\![\tau]\!]_\iota$. If we assume that variables initially hold some default value a^τ, then good-variable behaviour is stipulated as follows.

$$\gamma_\tau^\iota = B_\iota \cdot (read_\iota \cdot a_\iota^\tau \cdot B_\iota)^* \cdot \left(B_\iota \cdot \sum_{v \in \mathcal{A}[\![\tau]\!]} \big(write(v)_\iota \cdot ok \cdot B_\iota \cdot (read_\iota \cdot v_\iota \cdot B_\iota)^*\big) \right)^*$$

For the sake of completeness, $a^{\mathbf{int}} = 0$ and $a^{\mathbf{bool}} = \mathit{false}$. We can then give the semantics of blocks as

$$R_{\mathbf{new}[\tau]\ \iota\ \mathbf{in}\ M} = (\gamma_\iota^\tau \cap R_M) \mid_{\mathcal{A}[\![\tau]\!]_\iota} .$$

Note that the same intersection and hiding can be used to define $[\![\mathbf{new}[\tau]\ \iota\ \mathbf{in}\ M]\!]$ directly from $[\![M]\!]$: the bracketing moves, *run* and *done*, make no difference.

$$[\![\mathbf{new}[\tau]\ \iota\ \mathbf{in}\ M]\!] = (\gamma_\tau^\iota \cap [\![M]\!]) \mid_{\mathcal{A}[\![\tau]\!]_\iota} .$$

Theorem 2. Full abstraction. *Two terms of the recursion free second order finitary fragment of IA are equivalent (in full IA) if and only if the languages denoted by them are equal:*
For any $\Gamma \vdash P, Q : \theta, \qquad P \equiv Q \iff [\![P]\!] = [\![Q]\!]$.

Proof. We can show that the regular language denoted by a term of IA is equal to the set of complete plays in the fully abstract game semantics [1], therefore the full abstraction property is preserved. Note that language equivalence is asserted outside the fragment we describe here; witnesses to some inequivalences may belong to IA but not to the presented fragment. □

5 Examples of Reasoning

At this point a skeptical reader may entertain doubts concerning our earlier claim of simplicity. We have set up a formal notation of extended regular expressions which includes rather complicated operations. However, the complications are notational and not conceptual. Also, all the operations involved are defined effectively so carrying them out is a mechanical process. We hope that the simplicity of our approach will become clearer when we show examples of reasoning about putative equivalences.

Locality. This most simple of equivalences invalidates models of imperative computation relying on a global store, traceable back to Scott and Strachey [17]. It says that a globally defined procedure cannot modify a local variable, and it was first proved using the "possible worlds" model of Reynolds and Oles, constructed using functor categories [11].

$$P : \mathbf{comm} \vdash \ \mathbf{new}\ x\ \mathbf{in}\ P \equiv P$$

Proof.

$$\begin{aligned}
[\![\mathbf{new}\ x\ \mathbf{in}\ P]\!] &= (\gamma^x \cap [\![P]\!]) \restriction_{\mathcal{A}_x} \\
&= (\gamma^x \cap \mathit{run}\cdot \mathit{run}_P \cdot \mathit{done}_P \cdot \mathit{done}) \restriction_{\mathcal{A}_x} \\
&= (\mathit{run}\cdot \mathit{run}_P \cdot \mathit{done}_P \cdot \mathit{done}) \restriction_{\mathcal{A}_x} \\
&\quad \text{because no moves are tagged by } x \\
&= \mathit{run}\cdot \mathit{run}_P \cdot \mathit{done}_P \cdot \mathit{done} \\
&= [\![P]\!]
\end{aligned}$$

Snapback. This example captures the intuition that changes to the state are in some way irreversible. A procedure executing an argument which is a command inflicts upon the state changes that cannot be undone from within the procedure. This is why, in the following, if procedure P uses its argument both sides will fail to terminate; if procedure P does not use its argument the behaviour of each side will be identical because of the locality of x, as seen above. The first model to address this issue correctly was O'Hearn and Reynolds's interpretation of IA using the polymorphic linear lambda calculus [8]. Reddy also addressed this issue using a novel "object semantics" approach [13], but in a particular flavour of IA known as interference-controlled ALGOL [6]. A further development of this model, that also satisfies this equivalence, is O'Hearn and Reddy's [7], a model fully abstract for the second order subset.

$$\begin{aligned}
&P : \mathbf{comm} \to \mathbf{comm} \vdash \\
&\ \mathbf{new}\ x\ \mathbf{in}\ P(x := 1);\ \mathbf{if}\ !x = 1\ \mathbf{then}\ \Omega\ \mathbf{else}\ \mathbf{skip} \equiv P(\Omega)
\end{aligned}$$

Proof.

$$[\![x := 1]\!] = \mathit{run}\cdot \mathit{write}(1)_x \cdot \mathit{ok}_x \cdot \mathit{done}$$

$$[\![P(x := 1)]\!] = \mathit{run}\cdot \mathit{run}_P \cdot \left(\mathit{run}^1_P \cdot \mathit{write}(1)_x \cdot \mathit{ok}_x \cdot \mathit{done}^1_P\right)^* \cdot \mathit{done}_P \cdot \mathit{done}$$

$$[\![\mathbf{if}\ !x = 1\ \mathbf{then}\ \Omega\ \mathbf{else}\ \mathbf{skip}]\!] = \sum_{n \neq 1} \mathit{run}\cdot \mathit{read}_x \cdot n_x \cdot \mathit{done}$$

$$\begin{aligned}
&[\![P(x := 1);\ \mathbf{if}\ !x = 1\ \mathbf{then}\ \Omega\ \mathbf{else}\ \mathbf{skip}]\!] \\
&\quad = \mathit{run}\cdot \mathit{run}_P \cdot \left(\mathit{run}^1_P \cdot \mathit{write}(1)_x \cdot \mathit{ok}_x \cdot \mathit{done}^1_P\right)^* \cdot \mathit{done}_P \cdot \left(\sum_{n \neq 1} \mathit{read}_x \cdot n_x\right) \cdot \mathit{done}
\end{aligned}$$

$$\begin{aligned}
&\gamma^x \cap [\![P(x := 1);\ \mathbf{if}\ !x = 1\ \mathbf{then}\ \Omega\ \mathbf{else}\ \mathbf{skip}]\!] \\
&\quad = \mathit{run}\cdot \mathit{run}_P \cdot \mathit{done}_P \cdot \mathit{read}_x \cdot 0_x \cdot \mathit{done},
\end{aligned}$$

because the only possibility to complete a trace in $\sum_{n\neq 1} \mathit{read}_x \cdot n_x$ is if the trace in $\left(\mathit{run}^1_P \cdot \mathit{write}(1)_x \cdot \mathit{ok}_x \cdot \mathit{done}^1_P\right)^*$ is the empty trace. Otherwise, the good variable property of x requires $n_x = 1_x$, which is banned by the set to which n is restricted $(n \neq 1)$. The meaning of the left hand term of the equivalence is therefore:

$$\begin{aligned}&(\gamma^x \cap [\![P(x := 1);\ \mathbf{if}\ !x = 1\ \mathbf{then}\ \Omega\ \mathbf{else\ skip}]\!])\ |_{\mathcal{A}_x}\\&\quad = \mathit{run}\cdot \mathit{run}_P \cdot \mathit{done}_P \cdot \mathit{done} = [\![P(\Omega)]\!]\end{aligned}$$

Parametricity. The intuition of parametricity is one of representation independence. Procedures passed different but equivalent implementations of a data structure or algorithm are not supposed to be able to distinguish between them. Several such motivating examples are given by O'Hearn and Tennent [9], who introduce a model constructed using a certain relation-preserving functor category.

The specific example we give is of the equivalence of two implementations of a toggle-switch: one which uses 1 for "on" and −1 for "off", and one which uses **true** and **false**. The semantic equations for negation and the inequality test have not been spelled out but are the obvious ones.

$$\begin{aligned}&P : \mathbf{comm} \to \mathbf{exp}[\mathbf{bool}] \to \mathbf{comm} \vdash\\&\mathbf{new}[\mathbf{int}]\ x\ \mathbf{in}\ x := 1; P(x := -!x)(!x > 0)\\&\quad \equiv \mathbf{new}[\mathbf{bool}]\ x\ \mathbf{in}\ x := \mathbf{true}\ ; P(x := \mathbf{not}\ x)(!x)\end{aligned}$$

Proof.

$$[\![x := -!x]\!] = \sum_{n\in\mathcal{N}} \mathit{run}\cdot \mathit{read}_x \cdot n_x \cdot \mathit{write}(-n)_x \cdot \mathit{ok}_x \cdot \mathit{done}$$

$$[\![!x > 0]\!] = \sum_{n>0} q\cdot \mathit{read}_x \cdot n_x \cdot \mathit{true} + \sum_{n\leq 0} q\cdot \mathit{read}_x \cdot n_x \cdot \mathit{false}$$

$$\begin{aligned}&[\![x := 1; P(x := -!x)(!x > 0)]\!] = \mathit{run}\cdot \mathit{write}(1)_x \cdot \mathit{ok}_x \cdot\\&\quad \mathit{run}_P \cdot \left(\sum_{n\in\mathcal{N}} \mathit{run}^1_P \cdot \mathit{read}_x \cdot n_x \cdot \mathit{write}(-n)_x \cdot \mathit{ok}_x \cdot \mathit{done}^1_P +\right.\\&\quad \left.\sum_{n>0} q^2_P \cdot \mathit{read}_x \cdot n_x \cdot \mathit{true}^2_P + \sum_{n\leq 0} q^2_P \cdot \mathit{read}_x \cdot n_x \cdot \mathit{false}^2_P\right)^* \cdot \mathit{done}_P \cdot \mathit{done}\end{aligned}$$

$$\begin{aligned}&\gamma^x_{\mathbf{int}} \cap [\![x := 1; P(x := -!x)(!x > 0)]\!] =\\&\quad = \mathit{run}\cdot \mathit{write}(1)_x \cdot \mathit{ok}_x \cdot \mathit{run}_P \cdot (\epsilon + X + X\cdot Y + X\cdot Y\cdot X + \ldots)\cdot \mathit{done}_P \cdot \mathit{done}\\&\quad = \mathit{run}\cdot \mathit{write}(1)_x \cdot \mathit{ok}_x \cdot \mathit{run}_P \cdot \big(X + (X\cdot Y)^* \cdot (X + \epsilon)\big)\cdot \mathit{done}_P \cdot \mathit{done}\\&\quad \text{where } X = \mathit{run}^1_P \cdot \mathit{read}_x \cdot 1_x \cdot \mathit{write}(-1)_x \cdot \mathit{ok}_x \cdot \mathit{done}^1_P \cdot \left(q^2_P \cdot \mathit{read}_x \cdot (-1)_x \cdot \mathit{false}^2_P\right)^*\\&\quad \text{and } Y = \mathit{run}^1_P \cdot \mathit{read}_x \cdot (-1)_x \cdot \mathit{write}(1)_x \cdot \mathit{ok}_x \cdot \mathit{done}^1_P \cdot \left(q^2_P \cdot \mathit{read}_x \cdot (1)_x \cdot \mathit{true}^2_P\right)^*\end{aligned}$$

Why this is the case should be intuitively clear. A value of 1 is written into x, followed by negation only, which constrains all the plays to $(+1)_x$ and $(-1)_x$

only. The reads and writes have to match with the good variable behaviour. A fully formal proof is lengthier but trivial and mechanical. Restricting with $|_{\mathcal{A}[\![\mathbf{int}]\!]_x}$ gives the following trace for the left hand side:

$$run \cdot run_P \cdot \big(X' + (X' \cdot Y')^* \cdot (X' + \epsilon)\big) \cdot done_P \cdot done$$
$$\text{where } X' = run^1_P \cdot done^1_P \cdot \big(q^2_P \cdot false^2_P\big)^* \text{ and } Y' = run^1_P \cdot done^1_P \cdot \big(q^2_P \cdot true^2_P\big)^*$$

A similar calculation on the right hand side leads to the the same result.

6 Decidability and Complexity Issues

As we have seen, regular languages provide a semantics for the fragment of IA described here. To manipulate regular languages we have introduced a formal meta-language of extended regular expressions, which preserves regularity of the language. All the operations we have used in formulating the semantic valuations have been effectively given. Therefore, we can formulate the following obvious result:

Theorem 3 (Decidability). *Equivalence of two terms of the recursion free second order finitary fragment of IA is decidable.*

For the general problem of term equivalence the complexity bound appears to be at least of exponential space, as is the case for regular expressions with intersection [19]. However, the complexity bound for the general problem may not be relevant for the kind of terms that arise in the model of IA, and particularly for those that would be checked for equivalence in practice. This point, which will be investigated in future work, is of the utmost importance if a tool is to be developed based on our ideas.

Acknowledgments We are grateful to Robert Tennent, Samson Abramsky and Mark Jerrum for suggestions, questions and advice. We thank Peter O'Hearn and Pasquale Malacaria for a stimulating discussion on the paper, and our anonymous referees for their insightful comments.

References

1. S. Abramsky and G. McCusker. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions (extended abstract). In *Proceedings of 1996 Workshop on Linear Logic*, volume 3 of *Electronic notes in Theoretical Computer Science*. Elsevier, 1996. Also as Chapter 20 of [10].
2. S. Brookes. Full abstraction for a shared variable parallel language. In *Proceedings, 8th Annual IEEE Symposium on Logic in Computer Science*, pages 98–109, Montreal, Canada, 1993. IEEE Computer Society Press, Los Alamitos, California.
3. C. Hankin and P. Malacaria. Generalised flowcharts and games. *Lecture Notes in Computer Science*, 1443, 1998.
4. C. Hankin and P. Malacaria. A new approach to control flow analysis. *Lecture Notes in Computer Science*, 1383, 1998.

5. A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables: preliminary report. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 191–203, San Diego, California, 1988. ACM, New York. Reprinted as Chapter 7 of [10].
6. P. W. O'Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. Syntactic control of interference revisited. *Theoretical Computer Science*, 228:175–210, 1999. Preliminary version reprinted as Chapter 18 of [10].
7. P. W. O'Hearn and U. S. Reddy. Objects, interference and the Yoneda embedding. In S. Brookes, M. Main, A. Melton, and M. Mislove, editors, *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*, Tulane University, New Orleans, Louisiana, Mar. 29–Apr. 1 1995. Elsevier Science (http://www.elsevier.nl).
8. P. W. O'Hearn and J. C. Reynolds. From Algol to polymorphic linear lambda-calculus. Journal of the Association for Computing Machinery, to appear.
9. P. W. O'Hearn and R. D. Tennent. Relational parametricity and local variables. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–184, Charleston, South Carolina, 1993. ACM, New York. A version also published as Chapter 16 of [10].
10. P. W. O'Hearn and R. D. Tennent, editors. ALGOL-*like Languages.* Progress in Theoretical Computer Science. Birkhäuser, Boston, 1997. Two volumes.
11. F. J. Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages.* Ph.D. thesis, Syracuse University, Syracuse, N.Y., 1982.
12. A. M. Pitts. Reasoning about local variables with operationally-based logical relations. In *11th Annual Symposium on Logic in Computer Science*, pages 152–163. IEEE Computer Society Press, Washington, 1996. A version also published as Chapter 17 of [10].
13. U. S. Reddy. Global state considered unnecessary: Introduction to object-based semantics. LISP *and Symbolic Computation*, 9(1):7–76, 1996. Published also as Chapter 19 of [10].
14. J. C. Reynolds. Syntactic control of interference. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 39–46, Tucson, Arizona, Jan. 1978. ACM, New York.
15. J. C. Reynolds. The essence of ALGOL. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages,* Proceedings of the International Symposium on Algorithmic Languages, pages 345–372, Amsterdam, Oct. 1981. North-Holland, Amsterdam. Reprinted as Chapter 3 of [10].
16. D. A. Schmidt. On the need for a popular formal semantics. *ACM SIGPLAN Notices*, 32(1):115–116, Jan. 1997.
17. D. S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In J. Fox, editor, *Proceedings of the Symposium on Computers and Automata*, volume 21 of *Microwave Research Institute Symposia Series*, pages 19–46. Polytechnic Institute of Brooklyn Press, New York, 1971. Also Technical Monograph PRG-6, Oxford University Computing Laboratory, Programming Research Group, Oxford.
18. K. Sieber. Full abstraction for the second order subset of an ALGOL-like language. In *Mathematical Foundations of Computer Science*, volume 841 of *Lecture Notes in Computer Science*, pages 608–617, Kŏsice, Slovakia, Aug. 1994. Springer-Verlag, Berlin. A version also published as Chapter 15 of [10].
19. L. J. Stockmeyer. The complexity of decision problems in automata theory and logic. Technical Report MIT/LCS/TR-133, Massachusetts Institute of Technology, Laboratory for Computer Science, July 1974.