

The Regular-Language Semantics of First-Order Idealized ALGOL

Dan R. Ghica^a and Guy McCusker^b

^a*Department of Computing and Information Science, Queen's University,
Kingston, Ontario, K7L 3N6, Canada*

^b*School of Cognitive and Computing Sciences, University of Sussex at Brighton,
Falmer, Brighton, BN1 9QH, 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 model inherits its good theoretical properties; in fact, for first 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.

Key words: 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 [23]. 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 [18].

A common theme in the literature on semantics of IA, beginning with [13], 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 [19], relational parametricity was employed to capture representation-independence properties [17], and linear logic to explain irreversibility [16]. Recently, many of these ideas have been successfully incorporated in an operationally-based account of IA by Pitts [21].

A frustrating situation was created with the development of a fully abstract game semantics for IA [2]. 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 first-order subset of IA and its corresponding second-order equational theory, 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” [22] 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” [27]; 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]). 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 [8,9]. 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 a **let**-free language fragment. In the subsequent section we prove that this model is a correct representation of the games model. In section 6 we illustrate our model with several important putative equivalences. Section 7 address the issue of adding function-definitions. Other immediate syntactic and semantic extensions are discussed in the final section.

2 The programming language fragment

The principles of the programming language IA were laid down by John Reynolds in an influential paper [25]. 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 analyzed extensively [29,2,16]. We consider only the recursion-free first order fragment of this language, the fragment which has been used to give virtually all the significant equivalences mentioned in the literature, and its corresponding second-order equational theory. 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.

$$\begin{aligned} \tau &::= \mathbf{int} \mid \mathbf{bool}. \\ \sigma &::= \mathbf{comm} \mid \mathbf{var}\tau \mid \mathbf{exp}\tau, \\ \theta &::= \sigma \mid \sigma \times \sigma \times \cdots \times \sigma \rightarrow \sigma. \end{aligned}$$

Terms are introduced using type judgments of the form:

$$\Gamma \vdash M : \sigma, \quad \Gamma = \{\iota_1 : \theta_1, \dots, \iota_k : \theta_k\}.$$

We will be concerned with proving second-order equations of the form

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

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

$\overline{\Gamma \vdash \mathbf{skip} : \mathbf{comm}}$	$\overline{\Gamma \vdash \mathbf{diverge}_\sigma : \sigma}$
$\overline{\Gamma \vdash \mathbf{true} : \mathbf{expbool}}$	$\overline{\Gamma \vdash \mathbf{n} : \mathbf{expint}}$
$\overline{\Gamma, \iota : \theta \vdash \iota : \theta}$	$\frac{\Gamma \vdash V : \mathbf{var}\tau}{\Gamma \vdash !V : \mathbf{exp}\tau}$
$\frac{\Gamma \vdash E_1 : \mathbf{expint} \quad \Gamma \vdash E_2 : \mathbf{expint}}{\Gamma \vdash E_1 + E_2 : \mathbf{expint}}$	$\frac{\Gamma \vdash E_1 : \mathbf{expint} \quad \Gamma \vdash E_2 : \mathbf{expint}}{\Gamma \vdash E_1 = E_2 : \mathbf{expbool}}$
$\frac{\Gamma \vdash B_1 : \mathbf{expbool} \quad \Gamma \vdash B_2 : \mathbf{expbool}}{\Gamma \vdash B_1 \mathbf{and} B_2 : \mathbf{expbool}}$	$\frac{\Gamma \vdash B : \mathbf{expbool}}{\Gamma \vdash \mathbf{not} B : \mathbf{expbool}}$
$\frac{\Gamma \vdash V : \mathbf{var}\tau \quad \Gamma \vdash E : \mathbf{exp}\tau}{\Gamma \vdash V := E : \mathbf{comm}}$	$\frac{\Gamma \vdash C : \mathbf{comm} \quad \Gamma \vdash M : \sigma}{\Gamma \vdash C; M : \sigma}$
$\frac{\Gamma \vdash B : \mathbf{bool} \quad \Gamma \vdash M_1 : \sigma \quad \Gamma \vdash M_2 : \sigma}{\Gamma \vdash \mathbf{if} B \mathbf{then} M_1 \mathbf{else} M_2 : \sigma}$	$\frac{\Gamma \vdash B : \mathbf{expbool} \quad \Gamma \vdash C : \mathbf{comm}}{\Gamma \vdash \mathbf{while} B \mathbf{do} C : \mathbf{comm}}$
$\frac{\Gamma \vdash \iota : \sigma_1 \times \sigma_2 \times \dots \times \sigma_k \rightarrow \sigma \quad \Gamma \vdash M_i : \sigma_i}{\Gamma \vdash \iota(M_1, M_2, \dots, M_k) : \sigma}$	$\frac{\Gamma, \iota : \mathbf{var}\tau \vdash C : \mathbf{comm}}{\Gamma \vdash \mathbf{new}_\tau \iota \mathbf{in} C : \mathbf{comm}}$

Table 1

Terms and typing rules of IA

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 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**.

The language fragment as it stands is not appropriate as a programming lan-

guage in its own right, because although it contains terms with free identifiers of function type, it has no mechanism for binding such identifiers to functions. The fragment is sufficient for reasoning about program fragments, but not whole programs. This situation can be remedied by the addition of a function-definition constructor, with typing rule:

$$\frac{\Gamma, \iota : \sigma_0 \times \dots \times \sigma_m \rightarrow \sigma' \vdash Q : \sigma \quad \Gamma, \iota_0 : \sigma_0, \dots, \iota_m : \sigma_m \vdash P : \sigma'}{\Gamma \vdash \mathbf{let} \ \iota(\iota_0 : \sigma_0, \dots, \iota_m : \sigma_m) = P \ \mathbf{in} \ Q : \sigma},$$

yielding a self-contained programming language, which we shall call *first-order IA*. For the majority of this paper, we will concern ourselves only with the let-free fragment, since it admits a simple presentation of the semantics and contains all interesting examples from the IA literature. Furthermore, programs involving let can be reduced to programs without let by means of beta-reduction. However, for the sake of completeness, we consider the addition of let in section 7.

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 $\iota : \mathbf{expint} \rightarrow \mathbf{comm} \vdash M : \mathbf{comm}$, *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 \mathbf{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).

In more interesting games, such as the one used to interpret programs like $\iota : \mathbf{expint} \rightarrow \mathbf{comm} \vdash \iota(0) : \mathbf{comm}$, there are more moves. Corresponding to the result type \mathbf{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^{ι^1} , meaning “what is the value of the first argument to ι ?,” to which P may respond with an integer n , tagged as n^{ι^1} 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 (what is the first argument to ι?)
P: *0*¹ (the argument is 0)
O: *done*^ι (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*^ι 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.

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, 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 **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 ι(0), for example, is capable of supplying the argument 0 to ι 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, written in tabular format to indicate to what type occurrence every move belongs, together with the game for the constant pair (3, 5):

$$\begin{array}{ccc}
(3, 5) : \mathbf{expint} \times \mathbf{expint} & + : \mathbf{expint} \times \mathbf{expint} \longrightarrow \mathbf{expint} & \\
\begin{array}{c} q \\ 3 \end{array} & \begin{array}{c} q \\ m \end{array} & \begin{array}{c} q \\ n \end{array} \\
& \begin{array}{c} q \\ 5 \end{array} & \\
& & m + n
\end{array}$$

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

terms of the form $c : \mathbf{comm}, v : \mathbf{varint} \vdash M : \mathbf{comm}$, we may find interactions such as

$$run \cdot write(4)^v \cdot ok^v \cdot read^v \cdot 3^v \cdot run_c \cdot done_c \cdot read^v \cdot 7^v \dots$$

This play has two notable features:

- O has not played a good variable (in the sense of [24, 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 $reads$ yielded different values, although no explicit $writes$ 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) an *autonomous variable*.

Also, the interaction with v is not an observable part of the program’s behaviour. Therefore, the games interpretation of $\mathbf{new}\tau v \mathbf{in} M$ 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 [2], 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, $run \cdot run^t$ is not complete but $run \cdot run^t \cdot done^t \cdot done$ 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.*

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 *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 first-order types, this extra machinery is redundant. For a detailed account of the games model used here see [2,1,12].

4 The regular-language model

The programming language is interpreted using regular languages, denoted by *extended regular expressions*, defined as follows:

Definition 2 *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: $\emptyset, \epsilon \in \mathcal{R}_{\mathcal{A}}$; if $a \in \mathcal{A}$, then $a \in \mathcal{R}_{\mathcal{A}}$;

Iteration: if $R \in \mathcal{R}_{\mathcal{A}}$, $R^* \in \mathcal{R}_{\mathcal{A}}$;

Concatenation: if $R, S \in \mathcal{R}_{\mathcal{A}}$, then $R \cdot S \in \mathcal{R}_{\mathcal{A}}$;

Set operators: if $R, S \in \mathcal{R}_{\mathcal{A}}$, then $R + S, R \cap S, \overline{R} \in \mathcal{R}_{\mathcal{A}}$;

Restriction: if $R \in \mathcal{R}_{\mathcal{A}}$, $\mathcal{A}' \subseteq \mathcal{A}$, then $R|_{\mathcal{A}'} \in \mathcal{R}_{\mathcal{A} \setminus \mathcal{A}'}$,

$$R|_{\mathcal{A}'} = \{a_0 a_1 \cdots a_k \mid \exists w_i \in \mathcal{A}'^* \text{ s.t. } w_0 \cdot a_0 \cdot w_1 \cdots w_k \cdot a_k \cdot w_{k+1} \in R, a_i \in \mathcal{A} \setminus \mathcal{A}'\},$$

Substitution: if $R, S \in \mathcal{R}_{\mathcal{A}}$, $w \in \mathcal{A}^*$, then $R[w/S] \in \mathcal{R}_{\mathcal{A}}$,

$$R[w/S] = \sum_{y_0 \cdot w \cdot y_1 \cdots y_{k-1} \cdot w \cdot y_k \in R} y_0 \cdot S \cdot y_1 \cdots y_{k-1} \cdot S \cdot y_k, \quad y_i \in \mathcal{A}^*,$$

Broadening: if $\mathcal{B} \subseteq \mathcal{A}$, $R \in \mathcal{R}_{\mathcal{B}}$ then $\tilde{R} \in \mathcal{R}_{\mathcal{A}}$,

$$\tilde{R} = \sum_{a_0 a_1 \cdots a_n \in R} (\overline{\mathcal{B}})^* \cdot a_0 \cdot (\overline{\mathcal{B}})^* \cdot a_1 \cdot (\overline{\mathcal{B}})^* \cdots (\overline{\mathcal{B}})^* \cdot a_n \cdot (\overline{\mathcal{B}})^*, \quad a_i \in \mathcal{B}.$$

Constant \emptyset denotes the empty language, while ϵ is the language consisting only of the empty string. The constant a is the language of the singleton sequence a . Restriction represents the operation of removing from all sequences in a language the symbols from alphabet \mathcal{A}' . Substitutions represent the operation of substituting a new extended regular expression S for any occurrence of a given sub-sequence w in a regular language (regular language homomorphism). The broadening of a regular language over alphabet \mathcal{B} is obtained by introducing between consecutive symbols all strings from the complement of the language.

It is a routine exercise to show that:

Proposition 3 *Every extended regular expression $R \in \mathcal{R}_{\mathcal{A}}$ denotes a regular language.*

With every data-type τ and ground type σ we associate an alphabet $\mathcal{A} \llbracket - \rrbracket$:

$$\mathcal{A} \llbracket \text{int} \rrbracket = \mathcal{N} = \{n \mid -N < n < N\}, \quad \mathcal{A} \llbracket \text{bool} \rrbracket = \{tt, ff\}$$

$$\mathcal{A} \llbracket \text{comm} \rrbracket = \{run, done\},$$

$$\mathcal{A} \llbracket \text{exp} \tau \rrbracket = \{q, v \mid v \in \mathcal{A} \llbracket \tau \rrbracket\},$$

$$\mathcal{A} \llbracket \text{var} \tau \rrbracket = \{read, v, write(v), ok \mid v \in \mathcal{A} \llbracket \tau \rrbracket\}.$$

First-order types are interpreted using alphabets of the form:

$$\mathcal{A} \llbracket \sigma_1 \times \sigma_2 \times \cdots \times \sigma_k \rightarrow \sigma \rrbracket = \{a^i \mid a \in \mathcal{A} \llbracket \sigma_i \rrbracket, 1 \leq i \leq k\} \cup \mathcal{A} \llbracket \sigma \rrbracket,$$

where a^i is the result of the lexical operation of tagging symbol a with numeral i , resulting in a new symbol.

4.1 Basic term valuations

Interpretation $\llbracket - \rrbracket$ is provided for terms of the form $\Gamma \vdash M : \theta$, with Γ the type assignment for its free identifiers; the interpretation maps every term to a regular language. The regular-language interpretation is defined over the *context alphabet* of the term, defined as:

$$\mathcal{A}[\Gamma \vdash M : \theta] = \bigcup_{1 \leq j \leq k} \{a^{\iota_j} \mid a \in \mathcal{A}[\theta_j]\} \cup \bigcup_{1 \leq j \leq k} \mathcal{A}[\theta_j] \cup \mathcal{A}[\theta],$$

where $\Gamma = \{\iota_1 \mapsto \theta_1, \dots, \iota_k \mapsto \theta_k\}$ and a^{ι_j} is a new symbol resulting from the tagging of symbol a with identifier ι_j .

Every regular language that denotes the meaning of a term has a certain form, given by all its possible initial and final moves (*bracketing moves*). The bracketing moves merely indicate that a complete computation has occurred along with the outcome of the computation, but the actual interactions are given by the sub-expressions which form the meaning of the term:

$$\begin{aligned} \llbracket \Gamma \vdash M : \mathbf{comm} \rrbracket &= \mathit{run} \cdot \langle \Gamma \vdash M : \mathbf{comm} \rangle \cdot \mathit{done} \\ \llbracket \Gamma \vdash M : \mathbf{exp}\tau \rrbracket &= \sum_{n \in \mathcal{A}[\tau]} q \cdot \langle \Gamma \vdash M : \mathbf{exp}\tau \rangle_n \cdot n \\ \llbracket \Gamma \vdash M : \mathbf{var}\tau \rrbracket &= \sum_{n \in \mathcal{A}[\tau]} \mathit{read} \cdot \langle \Gamma \vdash M : \mathbf{exp}\tau \rangle_n^r \cdot n \\ &\quad + \sum_{n \in \mathcal{A}[\tau]} \mathit{write}(n) \cdot \langle \Gamma \vdash M : \mathbf{exp}\tau \rangle_n^w \cdot \mathit{ok}. \end{aligned}$$

Intuitively, a regular language $\langle \Gamma \vdash M : \mathbf{comm} \rangle$ is the actual computation carried out by M ; similarly, the regular language $\langle \Gamma \vdash M : \mathbf{exp}\tau \rangle_n$ is that particular computation, or play, in expression M which produces the value n . Finally, for variables, the regular language $\langle \Gamma \vdash M : \mathbf{exp}\tau \rangle_n^r$ describes what happens when value n is *read* from variable M while $\langle \Gamma \vdash M : \mathbf{exp}\tau \rangle_n^w$ describes the interactions associated with *writing* value n to that variable. If there is no danger of confusion we may only write $\llbracket M : \theta \rrbracket$ or $\langle M : \theta \rangle_n$ or even $\llbracket M \rrbracket$ or $\langle M \rangle_n$.

The regular expressions for arithmetic-logic constants, and their decompositions, are:

$$\begin{aligned} \langle n \rangle_n &= \epsilon, & \langle n \rangle_{m \neq n} &= \emptyset, \\ \langle \mathbf{true} \rangle_{\mathbf{true}} &= \epsilon, & \langle \mathbf{true} \rangle_{\mathbf{false}} &= \emptyset, \\ \langle \mathbf{false} \rangle_{\mathbf{false}} &= \epsilon, & \langle \mathbf{false} \rangle_{\mathbf{true}} &= \emptyset. \end{aligned}$$

Since the $\llbracket - \rrbracket$ and $\langle - \rangle$ regular languages can be recovered one from the other, the semantics of the language can be presented using either one. In the case of constants, the alternative formulation is:

$$\llbracket n \rrbracket_{\text{expint}} = q \cdot n \quad \llbracket \text{true} \rrbracket_{\text{expbool}} = q \cdot tt \quad \llbracket \text{false} \rrbracket_{\text{expbool}} = q \cdot ff.$$

We will tend to use $\langle - \rangle$ as often as possible, as it is the more concise formulation; but whenever needed we will revert to the $\llbracket - \rrbracket$ definitions.

$$\begin{aligned} \langle \text{not } B \rangle_b &= \langle B \rangle_{\text{not } b}, \quad b \in \{tt, ff\} \\ \langle E_1 \text{ op } E_2 \rangle_v &= \sum_{m, n \in \mathcal{A}[\tau], m \text{ op } n = v} \langle E_1 \rangle_m \cdot \langle E_2 \rangle_n, \quad v \in \mathcal{A}[\tau]. \end{aligned}$$

where **op** stands for any binary arithmetic-logic operator and *op* is its obvious interpretation. The interpretations for assignment and dereferencing are:

$$\begin{aligned} \langle V := E \rangle &= \sum_{v \in \mathcal{A}[\tau]} \langle E \rangle_v \cdot \langle V \rangle_v^w, \quad v \in \mathcal{A}[\tau], \\ \langle !V \rangle_v &= \langle V \rangle_v^r, \quad v \in \mathcal{A}[\tau]. \end{aligned}$$

The other imperative features (empty command, divergence, sequencing, iteration and branching) are:

$$\begin{aligned} \langle \text{skip} \rangle &= \epsilon, \\ \langle \text{diverge} \rangle &= \emptyset, \\ \langle C; C' \rangle &= \langle C \rangle \cdot \langle C' \rangle, \\ \langle C; E \rangle_v &= \langle C \rangle \cdot \langle E \rangle_v, \quad v \in \mathcal{A}[\tau] \\ \langle \text{while } B \text{ do } C \rangle &= \left(\langle B \rangle_{tt} \cdot \langle C \rangle \right)^* \cdot \langle B \rangle_{ff}, \\ \langle \text{if } B \text{ then } C \text{ else } C' \rangle &= \langle B \rangle_{tt} \cdot \langle C \rangle + \langle B \rangle_{ff} \cdot \langle C' \rangle. \end{aligned}$$

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 *p*-producing plays, where every such play is any concatenation of plays producing *m* and *n* in the arguments, if and only if $m \text{ op } n = p$. 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. This is exactly the traced-based interpretation used to interpret iteration as early as the '70s (see for example Section 2.3.4 in [10]). Non-termination **diverge** is interpreted as the empty set of complete plays.

Our semantics identifies divergence with the run-time error of numerical overflow, which comes about as a result of working with a finite set of integers.

Modeling the behaviour of arithmetical exceptions as **diverge** is an expedient approximation, coarse but not entirely unacceptable (see for example [26, Sections 2.7 and 5.1] for a discussion).

It may be helpful to illustrate the model with the proof of a simple fact:

Example 4 *The phrase $\Gamma \vdash \mathbf{while\ true\ do\ } C : \mathbf{comm}$ always diverges.*

PROOF. Applying the definitions, the meaning of the left-hand side is:

$$\begin{aligned} \llbracket \mathbf{while\ true\ do\ } C \rrbracket &= run \cdot (\llbracket \mathbf{true} \rrbracket_{tt} \cdot (\llbracket C \rrbracket)^* \cdot (\llbracket \mathbf{true} \rrbracket)_{ff} \cdot done \\ &= run \cdot (\epsilon \cdot (\llbracket C \rrbracket))^* \cdot \emptyset \cdot done = \emptyset = \llbracket \mathbf{diverge} \rrbracket \quad \blacksquare \end{aligned}$$

4.2 Free identifiers and application

We have already mentioned, in subsection 3.1, that in the game semantic model free identifiers are interpreted using a particular strategy, call the *copy-cat* strategy. Here we will introduce a regular language combinator which generates complete plays for a copy-cat strategy, along with the appropriate tagging. Tagging of a symbol with a string is a lexical operation, resulting in a new symbol.

Definition 5 *The copy-cat regular language $K_\sigma^{x/y}$, x and y strings, is:*

$$\begin{aligned} K_{\mathbf{comm}}^{x/y} &= run^x \cdot run^y \cdot done^y \cdot done^x, \\ K_{\mathbf{exp}\tau}^{x/y} &= \sum_{n \in \mathcal{A}[\tau]} q^x \cdot q^y \cdot n^y \cdot n^x, \\ K_{\mathbf{var}\tau}^{x/y} &= \sum_{n \in \mathcal{A}[\tau]} read^x \cdot read^y \cdot n^y \cdot n^x + \sum_{n \in \mathcal{A}[\tau]} write(n)^x \cdot write(n)^y \cdot ok^y \cdot ok^x. \end{aligned}$$

Free identifiers of ground and first-order types are interpreted as in Figure 1, where ‘ \bullet ’ is a special, reserved, symbol. For notational uniformity we will call these regular expressions $K_\theta^{\epsilon/\iota}$.

Application is interpreted by substituting the complete plays of the argument into the complete plays of the function:

$$\llbracket \iota(M_1, \dots, M_m) \rrbracket_\sigma = \llbracket \iota \rrbracket_{\sigma_1 \times \dots \times \sigma_k \rightarrow \sigma} [q^{\iota i} \cdot v^{\iota i} / (\llbracket M_i : \sigma_i \rrbracket_v)], \quad q, v \in \mathcal{A}[\sigma_i]$$

For example, let f be a free variable of type $\mathbf{comm} \rightarrow \mathbf{comm}$, and $M : \mathbf{comm}$. We then have:

$$\begin{aligned} \llbracket f(M) : \mathbf{comm} \rrbracket &= run \cdot run^f \cdot \left(run^{\bullet f^1} \cdot run^{f^1} \cdot done^{f^1} \cdot done^{\bullet f^1} \right)^* \cdot done^f \cdot done \\ &\quad \left[run^{f^1} \cdot done^{f^1} / (\llbracket M : \mathbf{comm} \rrbracket) \right] \\ &= run \cdot run^f \cdot \left(run^{\bullet f^1} \cdot (\llbracket M : \mathbf{comm} \rrbracket) \cdot done^{\bullet f^1} \right)^* \cdot done^f \cdot done \end{aligned}$$

$$\begin{aligned}
\llbracket \Gamma, \iota : \mathbf{comm} \vdash \iota : \mathbf{comm} \rrbracket &= K_{\mathbf{comm}}^{\epsilon/\iota} \\
\llbracket \Gamma, \iota : \mathbf{exp}\tau \vdash \iota : \mathbf{exp}\tau \rrbracket &= K_{\mathbf{exp}\tau}^{\epsilon/\iota} \\
\llbracket \Gamma, \iota : \mathbf{var}\tau \vdash \iota : \mathbf{var}\tau \rrbracket &= K_{\mathbf{var}\tau}^{\epsilon/\iota} \\
(\Gamma, \iota : \sigma_1 \times \cdots \times \sigma_k \rightarrow \mathbf{comm} \vdash \iota : \sigma_1 \times \cdots \times \sigma_k \rightarrow \mathbf{comm}) & \\
&= \mathit{run}^\iota \cdot (\sum_{1 \leq i \leq k} K_{\sigma_i}^{\bullet i/\iota})^* \cdot \mathit{done}^\iota \\
(\Gamma, \iota : \sigma_1 \times \cdots \times \sigma_k \rightarrow \mathbf{exp}\tau \vdash \iota : \sigma_1 \times \cdots \times \sigma_k \rightarrow \mathbf{exp}\tau) & \\
&= q^\iota \cdot (\sum_{1 \leq i \leq k} K_{\sigma_i}^{\bullet i/\iota})^* \cdot v^\iota, \quad v \in \mathcal{A}[\tau] \\
(\Gamma, \iota : \sigma_1 \times \cdots \times \sigma_k \rightarrow \mathbf{var}\tau \vdash \iota : \sigma_1 \times \cdots \times \sigma_k \rightarrow \mathbf{var}\tau) & \\
&= \mathit{read}^\iota \cdot (\sum_{1 \leq i \leq k} K_{\sigma_i}^{\bullet i/\iota})^* \cdot v^\iota, \quad v \in \mathcal{A}[\tau] \\
(\Gamma, \iota : \sigma_1 \times \cdots \times \sigma_k \rightarrow \mathbf{var}\tau \vdash \iota : \sigma_1 \times \cdots \times \sigma_k \rightarrow \mathbf{var}\tau) & \\
&= \mathit{write}(v)^\iota \cdot (\sum_{1 \leq i \leq k} K_{\sigma_i}^{\bullet i/\iota})^* \cdot \mathit{ok}^\iota, \quad v \in \mathcal{A}[\tau]
\end{aligned}$$

Fig. 1. Semantics of free identifiers

Intuitively, moves run^f and done^f are the unique effects caused by calling function f , while moves $\mathit{run}^{\bullet f^1}$ and $\mathit{done}^{\bullet f^1}$ are the effects caused by f whenever it evaluates its first, and in this case only, argument.

4.3 Local variables

The semantics of the local-variable block rely on the concept of *autonomous variable*, described earlier. It is simply the property of a variable behaving according to the expected causal rules of assignment and dereferencing. This behaviour can be encoded as a regular expression:

$$\gamma_\tau^\iota = (\mathit{read}^\iota \cdot v_\tau^\iota)^* \cdot \left(\sum_{v \in \mathcal{A}[\tau]} \mathit{write}(v)^\iota \cdot \mathit{ok}^\iota \cdot (\mathit{read}^\iota \cdot v^\iota)^* \right)^*,$$

where v_τ is the default value that a variable is initialized to, 0 for **int** and ff for **bool**. Initially, v_τ is the value read from the variable. Any legal play consists of a *write* followed by an arbitrary number of *reads*, all yielding the same value that was written. This series of moves can itself repeat indefinitely with different values.

The semantics of a local-variable block consist of two operations, imposing the good-variable behaviour on the local variable and removing all references to the variable, as it becomes invisible outside its binding scope. Intersection of regular languages is used to impose the first condition and restriction the second:

$$\llbracket \Gamma \vdash \mathbf{new}\tau \iota \mathbf{in} M : \mathbf{comm} \rrbracket = \left(\widetilde{\gamma}_\tau^\iota \cap \llbracket \Gamma, \iota : \mathbf{var}\tau \vdash M : \mathbf{comm} \rrbracket \right) \Big|_{\mathcal{A}_\tau^\iota},$$

where $\mathcal{A}_\tau^\iota = \{ \mathit{write}(v)^\iota, \mathit{ok}^\iota, \mathit{read}^\iota, v^\iota \mid v \in \llbracket \tau \rrbracket \}$. Notice that the regular ex-

pression γ_τ^ι is *broadened* so that only moves associated with variable ι are constrained by intersection.

4.4 Equational reasoning and full abstraction

Lemma 6 (Representation) *For any term $\Gamma \vdash M : \sigma$, the regular language $\llbracket \Gamma \vdash M : \sigma \rrbracket$ is the set of complete plays of the strategy interpreting it in the game semantics of [2], with justification pointers removed. Two terms $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M' : \sigma$ have the same regular language semantics if and only if the strategies they denote have the same set of complete plays.*

PROOF. The second part follows from the first once we observe that, at the types that we are considering, any sequence of moves can be properly justified in a unique way; this is immediate from the definitions of the game semantics.

The first part is proved by induction on the structure of terms, simply calculating using the two semantics. The details are in Section 5. ■

The regular-language model presented here is a representation of the games semantic model of the language fragment; therefore we expect the full abstraction property of the games semantic model to carry over to the regular language model.

Two sub-programs $P : \sigma$ and $Q : \sigma$ are called equivalent if they are inter-substitutable in any appropriate context without changing the meaning of the overall program; we denote two equivalent subprograms by $\Gamma \vdash P \equiv_\sigma Q$.

From the lemma and the full abstraction theorem in [2], it follows that:

Theorem 7 (Open full abstraction) *Two terms $\Gamma \vdash P : \sigma, Q : \sigma$ of the language fragment are equivalent if and only if their regular language interpretations are equal.*

$$\Gamma \vdash P \equiv_\sigma Q \quad \iff \quad \llbracket \Gamma \vdash P : \sigma \rrbracket = \llbracket \Gamma \vdash Q : \sigma \rrbracket.$$

It is important to point out that the full abstraction result is stated with respect to equivalence in the full IA language. Since the language fragment presented here does not include function definition, it can not be considered a stand-alone programming language; equivalence can not be defined with regard to it only, because it cannot provide enough discriminating contexts. First-order IA *qua* stand-alone programming language is discussed in Section 7.

From the full abstraction theorem and the representation lemma it follows directly that, since equality of regular languages is decidable:

Corollary 8 (Decidability) *Equivalence of subprograms of finitary, recursion free, first order IA with active expressions is decidable.*

5 Proof of the representation lemma

The reader who is not interested in the technical details of the proof of the Representation Lemma may skip this section without loss of continuity.

We introduce the following operation on languages:

Definition 9 *If S is a set of strings of form $m \cdot s_n \cdot n$, $m, n \in A^*$ and s_n does not use symbols from A , we define $S \mathbin{\text{\textcircled{;}}}_A R$ (read S composed with R over A) as $R[m \cdot n/s_n]$.*

When it is unambiguous, to simplify the notation, we may omit the subscript in $\mathbin{\text{\textcircled{;}}}_-$.

It is a routine exercise to show that:

Proposition 10 *If R and S are regular languages, for any set A if $S \mathbin{\text{\textcircled{;}}}_A R$ is well defined then it denotes a regular language.*

As a first step in proving the representation lemma we show that the interpretation of application is correct:

Proposition 11 *For any $\Gamma \vdash M : \sigma \rightarrow \theta, N : \sigma$,*

$$\llbracket \Gamma \vdash MN : \theta \rrbracket = \llbracket \Gamma \vdash N : \sigma \rrbracket \mathbin{\text{\textcircled{;}}}_{\mathcal{A}[\sigma]} \llbracket \Gamma \vdash M : \sigma \rightarrow \theta \rrbracket.$$

PROOF. The game semantic interpretation of application of $M : \sigma \rightarrow \theta$ to $N : \sigma$ is:

$$\llbracket \Gamma \rrbracket \xrightarrow{\langle \llbracket M \rrbracket, \llbracket N \rrbracket \rangle} \llbracket \sigma \rrbracket \Rightarrow \llbracket \theta \rrbracket \times \llbracket \sigma \rrbracket \xrightarrow{ev} \llbracket \theta \rrbracket$$

where ev is the *evaluation* strategy. This strategy in general does not have a regular set of complete plays, so the argument that $\llbracket MN \rrbracket = \llbracket M \rrbracket \mathbin{\text{\textcircled{;}}}_\sigma \llbracket N \rrbracket$ will be made directly at the level of the game semantics, by analyzing all the possible plays (Figure 2).

The opening question (q_θ^2) always occurs in $\llbracket \theta \rrbracket^2$, then is copied to $\llbracket \theta \rrbracket^1$ by ev (as q_θ^1). Subsequently, the strategy for $\llbracket M \rrbracket$ takes control of the play and it holds control of the play until a move (q_σ^1) occurs in $\llbracket \sigma \rrbracket^1$. This move transfers control back to ev , which copies it to $\llbracket \sigma \rrbracket^2$ (as q_σ^1). Subsequent play is then controlled by $\llbracket N \rrbracket$.

This is where the restriction to first order for M (ground type for N) insures that application is correctly represented by composition of regular expressions. Two observations are essential:

- (1) a next move to $\llbracket \theta \rrbracket^1$ is not possible because the play is governed by $\llbracket N \rrbracket$, which cannot use that type-component;
- (2) since σ is a ground type, $\llbracket N \rrbracket$ must complete its play after moves in $\llbracket \Gamma \rrbracket$ only. A higher order strategy would be able to ask a question in $\llbracket \sigma \rrbracket^2$, which ev would copy back to $\llbracket \sigma \rrbracket^1$, and give control back to $\llbracket M \rrbracket$. This

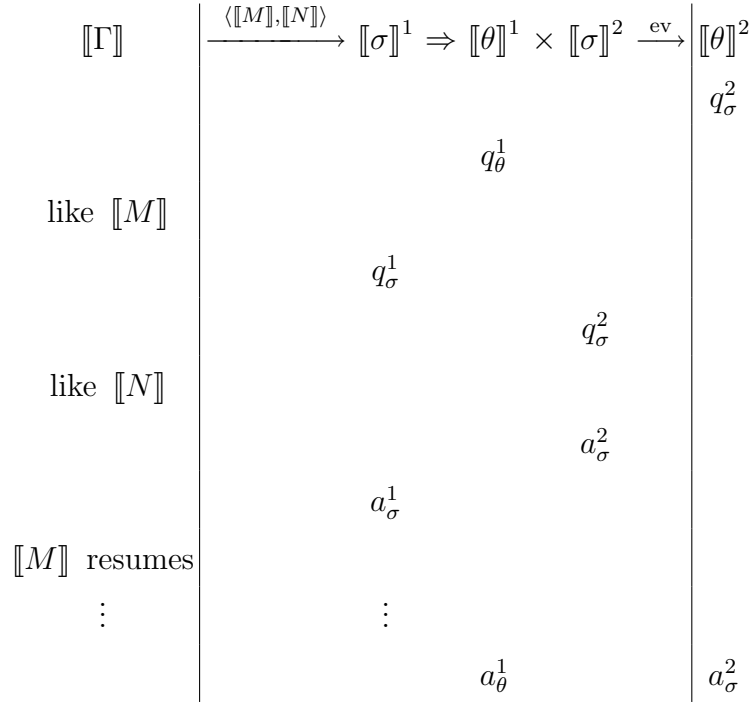


Fig. 2. Plays of function application

would cause a “nesting” of plays, rather than the simple interleaving of first order application.

Once $\llbracket N \rrbracket$ completes, the answer a_σ^2 is copied by ev from $\llbracket \sigma \rrbracket^2$ to $\llbracket \sigma \rrbracket^1$ and control switches back to $\llbracket M \rrbracket$. Because the justification pointer from a_σ^1 to q_σ^1 hides all play of $\llbracket N \rrbracket$ from $\llbracket M \rrbracket$, the latter simply resumes play from where it left off.

Finally, once $\llbracket M \rrbracket$ produces an answer a_θ^1 , it is relayed by ev to $\llbracket \theta \rrbracket^2$, closing the entire play.

We can see how the moves from the two σ components function as switches between the strategies of $\llbracket M \rrbracket$ and $\llbracket N \rrbracket$, inserting complete plays of $\llbracket N \rrbracket$ bracketed by q_σ^2 and a_σ^1 in the plays of $\llbracket M \rrbracket$, whenever moves $q_\sigma^1 \cdot a_\sigma^1$ occur.

Finally, all the moves from components $\llbracket \sigma \rrbracket^1 \Rightarrow \llbracket \theta \rrbracket^1 \times \llbracket \sigma \rrbracket^2$ are hidden, resulting in the same regular language as $\llbracket N \rrbracket \mathbin{\text{\textcircled{;}}}_\sigma \llbracket M \rrbracket$. ■

The interpretation of free identifiers is correct:

Proposition 12

$$\llbracket \iota : \theta \vdash \iota : \theta \rrbracket = K^{\epsilon/\iota}.$$

PROOF. For ground types the copy-cat strategy K is a finite set of strings. The equality can be verified by inspection.

$$\begin{aligned}
\llbracket + : \mathbf{expint}^1 \times \mathbf{expint}^2 \rightarrow \mathbf{expint}^3 \rrbracket &= \sum_{m,n \in N} q^3 \cdot q^1 \cdot m^1 \cdot q^2 \cdot n^2 \cdot (m+n)^3 \\
\llbracket - : \mathbf{expint}^1 \times \mathbf{expint}^2 \rightarrow \mathbf{expint}^3 \rrbracket &= \sum_{m,n \in N} q^3 \cdot q^1 \cdot m^1 \cdot q^2 \cdot n^2 \cdot (m-n)^3 \\
\llbracket \mathbf{or} : \mathbf{expbool}^1 \times \mathbf{expbool}^2 \rightarrow \mathbf{expbool}^3 \rrbracket &= \sum_{m,n \in \{tt,ff\}} q^3 \cdot q^1 \cdot m^1 \cdot q^2 \cdot n^2 \cdot (m \text{ or } n)^3 \\
\llbracket \mathbf{and} : \mathbf{expbool}^1 \times \mathbf{expbool}^2 \rightarrow \mathbf{expbool}^3 \rrbracket &= \sum_{m,n \in \{tt,ff\}} q^3 \cdot q^1 \cdot m^1 \cdot q^2 \cdot n^2 \cdot (m \text{ and } n)^3 \\
\llbracket = : \mathbf{exp}\tau^1 \times \mathbf{exp}\tau^2 \rightarrow \mathbf{exp}\tau^3 \rrbracket &= \sum_{m,n \in N} q^3 \cdot q^1 \cdot m^1 \cdot q^2 \cdot n^2 \cdot (m=n)^3 \\
\llbracket \mathbf{if} - \mathbf{then} - \mathbf{else} - : \mathbf{expbool}^1 \times \sigma^2 \times \sigma^3 \rightarrow \sigma^4 \rrbracket &= \\
&\sum_{\substack{a \in A_\sigma \\ q \in Q_\sigma}} q^4 \cdot q^1 \cdot tt^1 \cdot q^2 \cdot a^2 \cdot a^4 + \sum_{\substack{a \in A_\sigma \\ q \in Q_\sigma}} q^4 \cdot q^1 \cdot ff^1 \cdot q^3 \cdot a^3 \cdot a^4 \\
\llbracket - ; - : \mathbf{comm}^1 \times \sigma^2 \rightarrow \sigma^3 \rrbracket &= \sum_{\substack{a \in A_\sigma \\ q \in Q_\sigma}} q^3 \cdot run^1 \cdot done^1 \cdot q^2 \cdot a^2 \cdot a^3 \\
\llbracket - := - : \mathbf{var}\tau^1 \times \mathbf{exp}\tau^2 \rightarrow \mathbf{comm}^3 \rrbracket &= \\
&\sum_{a \in A[\tau]} run^3 \cdot q^2 \cdot a^2 \cdot write(a)^1 \cdot ok^1 \cdot done^3 \\
\llbracket ! : \mathbf{var}\tau^1 \rightarrow \mathbf{exp}\tau^2 \rrbracket &= \sum_{a \in A[\tau]} q^2 \cdot q^1 \cdot a^1 \cdot a^2
\end{aligned}$$

Fig. 3. Functional constants and operators

For first-order types the equality is immediate from the definitions of the games model, with the observation that for first order types the justification pointers are uniquely determined given a complete play. \blacksquare

Proposition 13 *The regular-language valuations (given in Section 4.1) for ground type constants, divergence, arithmetic-logic operators, assignment, deref-*

erencing, sequencing, branching and iteration are equal to the sets of complete plays of their game-semantic interpretation.

PROOF. For ground type constants, we can simply compare the regular-language interpretation with the games interpretation; their sets of complete plays are finite.

For divergence, by definition the set of complete plays is empty.

For a type σ let Q_σ and A_σ be the set of questions, respectively answers, in the games model. The sets of complete play for arithmetic-logic operators, assignment, dereferencing, sequencing and branching are given in Figure 3. They can be immediately compared with the game semantic definitions. The valuations, as given in Section 4.1, are arrived at using composition, for example:

$$\begin{aligned} \llbracket \Gamma \vdash M \mathbf{op} N : \sigma' \rrbracket &= \llbracket \Gamma \vdash \mathbf{op}(M, N) : \sigma' \rrbracket \\ &= \llbracket \Gamma \vdash M : \sigma^1 \rrbracket \mathbin{\dot{\circ}}_{\mathcal{A}[\sigma^1]} \left(\llbracket \Gamma \vdash N : \sigma^2 \rrbracket \mathbin{\dot{\circ}}_{\mathcal{A}[\sigma^2]} \llbracket \Gamma \vdash \mathbf{-op-} : \sigma^1 \times \sigma^2 \rightarrow \sigma' \rrbracket \right). \end{aligned}$$

Iteration requires a slightly more elaborate proof. The **while** construct is defined as syntactic sugar, using the recursion combinator:

$$\lambda m : \mathbf{comm}. \lambda b : \mathbf{expbool}. \mathbf{rec}[\mathbf{comm}](\lambda c : \mathbf{comm}. \mathbf{if} \ b \ \mathbf{then} \ m; c \ \mathbf{else} \ \mathbf{skip}).$$

But we have eliminated recursion from our language fragment, so we cannot use the meaning of **rec** in order to interpret **while**. We can however define it directly as:

$$\begin{aligned} W_\infty &= \bigcup_{i \in \mathbb{N}} W_i, \text{ where} \\ W_{i+1} &= W \mathbin{\dot{\circ}}_{\{\mathit{run}^c, \mathit{done}^c\}} W_i, \text{ for } i \geq 0, \ W_0 = \emptyset. \\ W &= \llbracket b : \mathbf{expbool}, m : \mathbf{comm}, c : \mathbf{comm} \vdash \mathbf{if} \ b \ \mathbf{then} \ m; c \ \mathbf{else} \ \mathbf{skip} : \mathbf{comm} \rrbracket \\ &= \mathit{run} \cdot q^b \cdot \mathit{tt}^b \cdot \mathit{run}^m \cdot \mathit{done}^m \cdot \mathit{run}^c \cdot \mathit{done}^c \cdot \mathit{done} + \mathit{run} \cdot q^b \cdot \mathit{ff}^b \cdot \mathit{done}. \end{aligned}$$

By mathematical induction on n it can be shown that in general:

$$W_n = \mathit{run} \cdot \sum_{i=0}^{n-1} (q^b \cdot \mathit{tt}^b \cdot \mathit{run}^m \cdot \mathit{done}^m)^i \cdot q^b \cdot \mathit{ff}^b \cdot \mathit{done}.$$

The superscript i indicates repeated concatenation. Then, it is easy to show that:

$$W_\infty = \bigcup_{i \in \mathbb{N}} W_i = \mathit{run}^3 \cdot (q^1 \cdot \mathit{tt}^1 \cdot \mathit{run}^2 \cdot \mathit{done}^2)^* \cdot q^1 \cdot \mathit{ff}^1 \cdot \mathit{done}^3,$$

This proves the Proposition for the game for **while**:

$$\begin{aligned} \llbracket \mathbf{while} - \mathbf{do} - : \mathbf{expbool}^1 \times \mathbf{comm}^2 \rightarrow \mathbf{comm}^3 \rrbracket \\ = \mathit{run}^3 \cdot (q^1 \cdot \mathit{tt}^1 \cdot \mathit{run}^2 \cdot \mathit{done}^2)^* \cdot q^1 \cdot \mathit{ff}^1 \cdot \mathit{done}^3. \end{aligned}$$

The formulation used in Section 4.1 is arrived at after composition. ■

The final language construct is **new**.

Proposition 14 *The regular-language $\llbracket \Gamma \vdash \mathbf{new}\tau \iota \mathbf{in} M : \mathbf{comm} \rrbracket$ is the set of complete plays of $\Gamma \vdash \mathbf{new}\tau \iota \mathbf{in} M : \mathbf{comm}$, in the game-semantic interpretation.*

PROOF. $\mathbf{new}\tau \iota \mathbf{in} M$ is syntactic sugar for $\mathbf{new}(\lambda\iota:\mathbf{var}\tau.M)$, where $\mathbf{new} : (\mathbf{var}\tau \rightarrow \mathbf{comm}) \rightarrow \mathbf{comm}$. According to the game-semantic definition, a complete play of **new** is of the form:

$$\begin{array}{c} (\mathbf{var}\tau \rightarrow \mathbf{comm}) \rightarrow \mathbf{comm} \\ \mathit{run} \\ \mathit{run} \\ S \\ \mathit{done} \\ \mathit{done} \end{array}$$

where S is any string in γ_τ^ι . Using the game-semantic definition of composition and an analysis of possible moves similar to the one in Proposition 11, it follows that the complete plays of $\mathbf{new}(\lambda\iota:\mathbf{var}\tau.M)$ are $(\widetilde{\gamma}_\tau^\iota \cap \llbracket \Gamma, \iota:\mathbf{var}\tau \vdash M \rrbracket) \upharpoonright_{A_\tau^\iota}$. ■

This concludes the proof of the Representation Lemma.

6 Example equations

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.

6.1 Call-by-name

It is typical of call-by name languages to have *non-strict* procedures, that is procedures which may not use their argument at all. Therefore the following inequivalence is true:

$$f : \mathbf{comm} \rightarrow \mathbf{comm} \vdash f(\mathbf{diverge}_{\mathbf{comm}}) \not\equiv_{\mathbf{comm}} \mathbf{diverge}_{\mathbf{comm}}.$$

PROOF.

$$\begin{aligned} & \llbracket f(\mathbf{diverge}) \rrbracket \\ &= \left(\mathit{run} \cdot \mathit{run}^f \cdot (\mathit{run}^{\bullet f^1} \cdot \mathit{run}^{f^1} \cdot \mathit{done}^{f^1} \cdot \mathit{done}^{\bullet f^1})^* \cdot \mathit{done}^f \cdot \mathit{done} \right) [\mathit{run}^{f^1} \cdot \mathit{done}^{f^1} / \emptyset] \\ &= \mathit{run} \cdot \mathit{run}^f \cdot (\emptyset)^* \cdot \mathit{done}^f \cdot \mathit{done} = \mathit{run} \cdot \mathit{run}^f \cdot \epsilon \cdot \mathit{done}^f \cdot \mathit{done} \\ &= \mathit{run} \cdot \mathit{run}^f \cdot \mathit{done}^f \cdot \mathit{done} \neq \emptyset = \llbracket \mathbf{diverge}_{\mathbf{comm}} \rrbracket \quad \blacksquare \end{aligned}$$

6.2 Locality

This deceptively simple equivalence shown below is not validated by the traditional models of imperative computation relying on a global store model, traceable back to Scott and Strachey [28]. It reflects the fact that a non-locally defined procedure cannot modify a local variable, and it was first proved in the “possible worlds” model of Reynolds and Oles, constructed using functor categories [19]:

$$p : \mathbf{comm} \vdash \mathbf{new} \tau \ x \ \mathbf{in} \ p \equiv p.$$

PROOF.

$$\begin{aligned} \llbracket \mathbf{new} \tau \ x \ \mathbf{in} \ p \rrbracket &= \widetilde{\gamma}_\tau^x \cap \llbracket p \rrbracket \Big|_{\mathcal{A}_\tau^x} = (\widetilde{\gamma}_\tau^x \cap \mathit{run} \cdot \mathit{run}^p \cdot \mathit{done}^p \cdot \mathit{done}) \Big|_{\mathcal{A}_\tau^x} \\ &= (\mathit{run} \cdot \mathit{run}^p \cdot \mathit{done}^p \cdot \mathit{done}) \Big|_{\mathcal{A}_\tau^x} = \mathit{run} \cdot \mathit{run}^p \cdot \mathit{done}^p \cdot \mathit{done} = \llbracket p \rrbracket \quad \blacksquare \end{aligned}$$

6.3 Snapback

This example 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 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 correctly address this issue was O’Hearn and Reynolds’s interpretation of IA using the polymorphic linear lambda calculus [16]. Reddy

also addressed this issue using a novel “object semantics” approach [22], but in a particular flavour of IA known as interference-controlled ALGOL [14]. A further development of the model, which also satisfies this equivalence, is O’Hearn and Reddy’s [15], a model fully abstract for the second order subset.

$$\begin{aligned} p : \mathbf{comm} &\rightarrow \mathbf{comm} \vdash \\ &\mathbf{new}_\tau x \text{ in } p(x := 1); \mathbf{if } !x = 1 \text{ then } \mathbf{diverge} \equiv p(\mathbf{diverge}), \end{aligned}$$

where $\mathbf{if } B \text{ then } C$ stands for $\mathbf{if } B \text{ then } C \text{ else } \mathbf{skip}$.

PROOF.

$$\begin{aligned} &\llbracket \mathbf{new}_\tau x \text{ in } p(x := 1); \mathbf{if } !x = 1 \text{ then } \mathbf{diverge} \rrbracket \\ &= \left(\widetilde{\gamma}_\tau^x \cap \llbracket p(x := 1); \mathbf{if } !x = 1 \text{ then } \mathbf{diverge} \rrbracket \right) \Big|_{\mathcal{A}_\tau^x}. \end{aligned}$$

Intermediate calculations give:

$$\llbracket p(x := 1) \rrbracket = \mathit{run} \cdot \mathit{run}^p \cdot \left(\mathit{run}^{\bullet p1} \cdot \mathit{write}^x(1) \cdot \mathit{ok}^x \cdot \mathit{done}^{\bullet p1} \right)^* \cdot \mathit{done}^p \cdot \mathit{done}$$

and:

$$\llbracket \mathbf{if } !x = 1 \text{ then } \mathbf{diverge} \rrbracket = \sum_{n \neq 1} \mathit{run} \cdot \mathit{read}^x \cdot n^x \cdot \mathit{done}$$

Therefore, the interpretation of the left-hand side is the regular language:

$$\begin{aligned} &\widetilde{\gamma}_\tau^x \cap \mathit{run} \cdot \mathit{run}^p \cdot \left(\mathit{run}^{\bullet p1} \cdot \mathit{write}^x(1) \cdot \mathit{ok}^x \cdot \mathit{done}^{\bullet p1} \right)^* \cdot \mathit{done}^p \\ &\quad \cdot \left(\sum_{n \neq 1} \mathit{read}^x \cdot n^x \right) \cdot \mathit{done} \Big|_{\mathcal{A}_\tau^x}. \end{aligned}$$

Notice that imposing the autonomous-variable behaviour on any play of the form $\dots \mathit{write}^x(1) \cdot \mathit{ok}^x \dots \mathit{read}^x \cdot n^x \dots$ requires $n = 1$. But the value $n = 1$ is not part of the possible plays of $\sum_{n \neq 1} \mathit{read}^x \cdot n^x$, so the moves $\mathit{write}^x(1) \cdot \mathit{ok}^x$ must be excluded altogether in order to complete the full play. This implies that the iteration is empty, *i.e.* procedure p does not use its arguments. After eliminating all moves in \mathcal{A}_τ^x the left hand side reduces to:

$$\mathit{run} \cdot \mathit{run}^p \cdot \mathit{done}^p \cdot \mathit{done},$$

which is the same as the right-hand side (see subsection 6.1). ■

6.4 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 [17], 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**.

$$\begin{aligned} p : \mathbf{comm} \times \mathbf{expbool} &\rightarrow \mathbf{comm} \vdash \\ \mathbf{newint} \ x \ \mathbf{in} \ x := 1; p(x := -!x, !x > 0) \\ &\equiv \mathbf{newbool} \ x \ \mathbf{in} \ x := \mathbf{true}; p(x := \mathbf{not} \ x, !x) \end{aligned}$$

PROOF.

$$\begin{aligned} \llbracket \mathbf{newint} \ x \ \mathbf{in} \ x := 1; p(x := -!x, !x > 0) \rrbracket \\ = \widetilde{\gamma_{\mathbf{bool}}^x} \cap \llbracket x := 1; p(x := -!x, !x > 0) \rrbracket. \end{aligned}$$

Intermediate calculations yield:

$$\begin{aligned} \llbracket x := 1; p(x := -!x, !x > 0) \rrbracket \\ = \mathit{run} \cdot \mathit{write}^x(1) \cdot \mathit{ok}^x \cdot \mathit{run}^p \cdot \left(\mathit{run}^{\bullet p1} \cdot \sum_{n \in \mathcal{N}} \mathit{read}^x \cdot n^x \cdot \mathit{write}^x(-n) \cdot \mathit{ok}^x \cdot \mathit{done}^{\bullet p1} \right. \\ \left. + q^{\bullet p2} \cdot \sum_{n > 0} \mathit{read}^x \cdot n^x \cdot \mathit{tt}^{\bullet p2} + q^{\bullet p2} \cdot \sum_{n \leq 0} \mathit{read}^x \cdot n^x \cdot \mathit{ff}^{\bullet p2} \right)^* \cdot \mathit{done}^p \cdot \mathit{done}. \end{aligned}$$

Imposing the autonomous-variable constraint $\gamma_{\mathbf{bool}}^x$ on this regular expression produces the regular expression:

$$\mathit{run} \cdot \mathit{write}^x(1) \cdot \mathit{ok}^x \cdot \mathit{run}^p \cdot \left((S \cdot U)^* \cdot T^* + (S \cdot U)^* \cdot S \cdot F^* \right)^* \cdot \mathit{done}^p \cdot \mathit{done},$$

where

$$\begin{aligned} S &= \mathit{run}^{\bullet p1} \cdot \mathit{read}^x \cdot 1^x \cdot \mathit{write}^x(-1) \cdot \mathit{ok}^x \cdot \mathit{done}^{\bullet p1}, \\ U &= \mathit{run}^{\bullet p1} \cdot \mathit{read}^x \cdot (-1)^x \cdot \mathit{write}^x(1) \cdot \mathit{ok}^x \cdot \mathit{done}^{\bullet p1}, \\ T &= q^{\bullet p2} \cdot \mathit{read}^x \cdot 1^x \cdot \mathit{tt}^{\bullet p2}, \\ F &= q^{\bullet p2} \cdot \mathit{read}^x \cdot (-1)^x \cdot \mathit{ff}^{\bullet p2}. \end{aligned}$$

Finally restricting with $\mathcal{A}_{\mathbf{bool}}^x$ gives the interpretation for the right-hand side:

$$\begin{aligned} \mathit{run} \cdot \mathit{run}^p \cdot \left((\mathit{run}^{\bullet p1} \cdot \mathit{done}^{\bullet p1} \cdot \mathit{run}^{\bullet p1} \cdot \mathit{done}^{\bullet p1})^* \cdot (q^{\bullet p2} \cdot \mathit{tt}^{\bullet p2})^* + \right. \\ \left. (\mathit{run}^{\bullet p1} \cdot \mathit{done}^{\bullet p1} \cdot \mathit{run}^{\bullet p1} \cdot \mathit{done}^{\bullet p1})^* \cdot \mathit{run}^{\bullet p1} \cdot \mathit{done}^{\bullet p1} \cdot (q^{\bullet p2} \cdot \mathit{ff}^{\bullet p2})^* \right)^* \cdot \mathit{done}^p \cdot \mathit{done}. \end{aligned}$$

The intuitive interpretation is that if procedure p uses its first argument an even number of times then the second argument evaluates to true; if it uses its

first argument an odd number of times then the second argument evaluates to false. This captures the dynamic behaviour of the switch-like procedure.

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

6.5 Consequences of restriction to finite data set

In [17] there is another equivalence which tests the parametric model, involving two quite different implementation of an “irreversible” switch:

$$\begin{aligned} p : \mathbf{comm} \times \mathbf{expbool} \rightarrow \mathbf{comm} \vdash \\ \mathbf{newint} \ x \ \mathbf{in} \ x := 0; p(x := !x + 1, !x > 0) \\ \equiv \mathbf{newbool} \ x \ \mathbf{in} \ x := 0; p(x := 1, !x > 0). \end{aligned}$$

The implementation on the left achieves the “on” position by incrementing x , while the implementation on the right achieves the same effect by assigning 1 to x . From within the procedure p the effect is the same—but with one important caveat. It must be possible for variable x to be increased indefinitely, that is the type **int** must not be finite. Using the first argument N times, where N is the upper bound of the finite subset of integers, causes overflow (and consequently divergence) on the left, but not on the right. Our model therefore does not validate this equivalence.

Also, certain equivalences which are invalid with an infinite dataset become valid in a finitary fragment. For example, if the data-set is only $\{-1, 0, 1\}$ the following equivalence holds:

$$\begin{aligned} \mathbf{newint} \ x \ \mathbf{in} \ x := E; \ \mathbf{if} \ !x = -1 \ \mathbf{or} \ !x = 0 \ \mathbf{or} \ !x = 1 \ \mathbf{then} \ \mathbf{diverge} \\ \equiv \mathbf{diverge}. \end{aligned}$$

But this is not showing a weakness in the model itself. It is unsurprising that the properties of a language with an infinite data-set change when only its finitary fragment is considered.

7 Functions

In order to achieve a stand-alone programming language, we now show how function definitions can be added to the language fragment. We will be able to show that the regular language model is fully abstract without a need to qualify the full abstractness result in the larger context of full IA.

The syntax of function definition is:

$$\frac{\Gamma, \iota : \sigma_0 \times \dots \times \sigma_m \rightarrow \sigma' \vdash Q : \sigma \quad \Gamma, \iota_0 : \sigma_0, \dots, \iota_m : \sigma_m P : \sigma'}{\Gamma \vdash \mathbf{let} \ \iota(\iota_0 : \sigma_0, \dots, \iota_m : \sigma_m) = P \ \mathbf{in} \ Q : \sigma}.$$

First order functions are imposed uniformly over ground types, including variables. Function calls can be therefore used as l-values (as in C++) and are arbitrarily nested (as in PASCAL).

Semantically, the most straightforward way to handle this addition is to depart even further from the game semantic model and introduce an *environment* as a parameter to the semantic valuation function; the semantics of a term is given by $\llbracket M \rrbracket_\theta u$, where the environment u maps identifiers to regular languages, $u : \text{dom}(\Gamma) \rightarrow R_{\mathcal{A}}$.

Function definition (**let**) is modeled by associating *copy-cat* regular expressions with the formal parameters in the environment:

$$\begin{aligned} \llbracket \mathbf{let} \ \iota(\iota_0 : \sigma_0, \dots, \iota_m : \sigma_m) = P \ \mathbf{in} \ Q \rrbracket_\sigma u \\ = \llbracket Q \rrbracket_\sigma \left(u \mid \iota \mapsto \llbracket P \rrbracket_{\sigma'} (u \mid \iota_0 \mapsto K_{\sigma_0}^{\epsilon/\iota_0} \mid \dots \mid \iota_m \mapsto K_{\sigma_m}^{\epsilon/\iota_m}) \right). \end{aligned}$$

The new interpretation of identifiers is:

$$\llbracket \iota \rrbracket_\theta u = u(\iota),$$

and the interpretation of local variables:

$$\llbracket \mathbf{new}_\tau \ \iota \ \mathbf{in} \ M \rrbracket_{\mathbf{comm}} u = \left(\widetilde{\gamma}_\tau^\iota \cap \llbracket M \rrbracket (u \mid \iota \mapsto K_{\mathbf{var}_\tau}^{\epsilon/\iota}) \right) \Big|_{\mathcal{A}_\tau}.$$

The rest of the language constructs do not need to use the environment, so their semantic valuations are similar to the old ones (Table 2).

An initial environment u_0 is used to give meaning to the free identifiers. In the full IA, **let** can be considered as syntactic sugar for an appropriate lambda term:

$$\begin{aligned} \mathbf{let} \ \iota(\iota_0 : \sigma_0, \dots, \iota_n : \sigma_n) = P \ \mathbf{in} \ Q \\ = (\lambda \iota : \sigma_0 \times \dots \times \sigma_n. Q)(\lambda \iota_0 : \sigma_0. \dots \lambda \iota_n : \sigma_n. P), \end{aligned}$$

for all occurrences of $\iota(M_0, \dots, M_n)$ in Q . We state the following representation lemma, relating the game semantics, as presented in the previous sections, with this environment-based semantics.

Lemma 15 (Representation)

$$\llbracket P \rrbracket_\sigma u_0 = \llbracket \Gamma \vdash P : \sigma \rrbracket^{\mathit{game}},$$

where $\Gamma = \iota_0 : \theta_0, \dots, \iota_n : \theta_n$ and $u_0 = \{ \iota_0 \mapsto K_{\theta_0}^{\epsilon/\iota_0}, \dots, \iota_n \mapsto K_{\theta_n}^{\epsilon/\iota_n} \}$.

PROOF. We will show that $\llbracket M \rrbracket_\theta u_0 = \llbracket \Gamma \vdash M \rrbracket_\theta^{\mathit{game}}$ by structural induction on the syntax of M .

For all let-free term constructors the proof follows directly from the definition of the semantics and the fact that the initial environment is stipulated to map

$$\begin{aligned}
\langle \mathbf{k} \rangle_k u &= \epsilon & \langle \mathbf{k} \rangle_{k' \neq k} u &= \emptyset, & \mathbf{k} & \text{any boolean or numeric constant} \\
\langle \mathbf{not } B \rangle_b u &= \langle B \rangle_{\mathbf{not } b} u, & b &\in \{tt, ff\} \\
\langle E_1 \mathbf{op} E_2 \rangle_v u &= \sum_{m, n \in \mathcal{A}[\tau], m \mathbf{op} n = v} \langle E_1 \rangle_m u \cdot \langle E_2 \rangle_n u, & v &\in \mathcal{A}[\tau] \\
\langle V := E \rangle u &= \sum_{v \in \mathcal{A}} \llbracket E \rrbracket_v u \cdot \langle V \rangle_v^w u \\
\langle !V \rangle_v u &= \langle V \rangle_v^r u, & v &\in \mathcal{A}[\tau] \\
\langle \mathbf{skip} \rangle u &= \epsilon & \langle \mathbf{diverge} \rangle u &= \emptyset \\
\langle C; C' \rangle u &= \langle C \rangle u \cdot \langle C' \rangle u \\
\langle C; E \rangle_v u &= \langle C \rangle u \cdot \langle E \rangle_v u, & v &\in \mathcal{A}[\tau] \\
\langle \mathbf{while } B \mathbf{do} C \rangle u &= (\langle B \rangle_{tt} u \cdot \langle C \rangle u)^* \cdot \langle B \rangle_{ff} u \\
\langle \mathbf{if } B \mathbf{then } C \mathbf{else } C' \rangle u &= \langle B \rangle_{tt} u \cdot \langle C \rangle u + \langle B \rangle_{ff} u \cdot \langle C' \rangle u \\
\llbracket \iota(M_1, \dots, M_m) \rrbracket u &= (\llbracket \iota \rrbracket u)[w^{\iota_i} \cdot v^{\iota_i} / \langle M_i \rangle_v u], & w, v &\in \mathcal{A}[\sigma_i], M_i : \sigma_i
\end{aligned}$$

where

$$\begin{aligned}
\llbracket M \rrbracket_{\mathbf{comm}} u &= \mathit{run} \cdot \langle M \rangle u \cdot \mathit{done} \\
\llbracket M \rrbracket_{\mathbf{exp}\tau} u &= \sum_{n \in \mathcal{A}[\tau]} q \cdot \langle M \rangle_n u \cdot n \\
\llbracket M \rrbracket_{\mathbf{var}\tau} u &= \sum_{n \in \mathcal{A}[\tau]} \mathit{read} \cdot \langle M \rangle_n^r u \cdot n \\
&\quad + \sum_{n \in \mathcal{A}[\tau]} \mathit{write}(n) \cdot \langle M \rangle_n^w u \cdot \mathit{ok}
\end{aligned}$$

Table 2

Environment-based valuations

free identifiers to the same regular languages (copy-cat) as their interpretation in the semantics as previously presented.

We shall use the fact that:

$$\begin{aligned}
&(\lambda \iota : \sigma_0 \times \dots \times \sigma_n. Q)(\lambda \iota_0 : \sigma_0. \dots \lambda \iota_n : \sigma_n. P) \\
&= Q[\lambda \iota_0 : \sigma_0. \dots \lambda \iota_n : \sigma_n. P / \iota] \\
&= Q[P[M_i / \iota_i] / \iota(M_0, \dots, M_n)].
\end{aligned}$$

The first step is β -reduction; the last step holds because syntactically the only construction an identifier denoting a function can be involved is application. Using the above, we need to show that:

$$\llbracket \mathbf{let } \iota(\iota_0 : \sigma_0, \dots, \iota_n : \sigma_n) = P \mathbf{in} Q \rrbracket_\theta u_0 = \left[\Gamma \vdash Q[P[M_i / \iota_i] / \iota(M_0, \dots, M_n)] \right]_\theta^{\mathbf{game}},$$

But:

$$\begin{aligned}
& \llbracket \mathbf{let} \ \iota(\iota_0:\sigma_0, \dots, \iota_n:\sigma_n) = P \ \mathbf{in} \ Q \rrbracket_\theta u_0 \\
&= \llbracket Q \rrbracket (u_0 \mid \iota \mapsto \llbracket P \rrbracket (u_0 \mid \iota_0 \mapsto K_{\sigma_0}^{\epsilon/\iota_0})) \\
&= \llbracket Q \rrbracket (u_0 \mid \iota \mapsto \llbracket P \rrbracket u'_0),
\end{aligned}$$

where u'_0 is the initial environment of P , mapping all its free identifiers to copy-cat regular expressions. Applying the induction hypothesis on P it follows that:

$$\llbracket \mathbf{let} \ \iota(\iota_0:\sigma_0, \dots, \iota_n:\sigma_n) = P \ \mathbf{in} \ Q \rrbracket_\theta u_0 = \llbracket Q \rrbracket (u_0 \mid \iota \mapsto \llbracket \Gamma, \iota_0:\sigma_0 \vdash P \rrbracket^{\text{game}}),$$

so, equivalently, what we need to prove is that:

$$\llbracket Q \rrbracket (u_0 \mid \iota \mapsto \llbracket \Gamma, \iota_0:\sigma_0 \vdash P \rrbracket^{\text{game}}) = \llbracket \Gamma \vdash Q [P[M_i/\iota_i]/\iota(M_0, \dots, M_n)] \rrbracket_\theta^{\text{game}}$$

We do this by structural induction on the syntax of Q .

For constants and identifiers the proof is immediate.

For the combinators of the language that do not involve function binding (assignment, arithmetic-logic operators, sequencing, branching, iteration, local variable) the proofs are similar; we will show only sequential composition in detail. Without loss of generality, assume ι only has one argument.

$$\begin{aligned}
& \llbracket C; C' \rrbracket (u_0 \mid \iota \mapsto \llbracket \Gamma, \iota_0:\sigma_0 \vdash P \rrbracket^{\text{game}}) \\
&= \mathit{run} \cdot \llbracket C \rrbracket (u_0 \mid \iota \mapsto \llbracket \Gamma, \iota_0:\sigma_0 \vdash P \rrbracket^{\text{game}}) \\
&\quad \cdot \llbracket C' \rrbracket (u_0 \mid \iota \mapsto \llbracket \Gamma, \iota_0:\sigma_0 \vdash P \rrbracket^{\text{game}}) \cdot \mathit{done}.
\end{aligned}$$

Applying the induction hypothesis on the two let-terms, this further equals:

$$\begin{aligned}
& \mathit{run} \cdot \left(\llbracket \Gamma \vdash C [P[M_1/\iota_0]/\iota M_1] \rrbracket^{\text{game}} \cdot \left(\llbracket \Gamma \vdash C' [P[M_2/\iota_0]/\iota M_2] \rrbracket^{\text{game}} \cdot \mathit{done} \right) \right) \\
&= \llbracket \llbracket \Gamma \vdash C [P[M_1/\iota_0]/\iota M_1]; C' [P[M_2/\iota_0]/\iota M_2] \rrbracket^{\text{game}} \\
&= \llbracket \llbracket \Gamma \vdash (C; C') [P[M/\iota_0]/\iota M] \rrbracket^{\text{game}},
\end{aligned}$$

for all occurrences of ιM_1 in C , ιM_2 in C' , ιM in $C; C'$.

For application:

$$\begin{aligned}
& \llbracket l'Q \rrbracket (u_0 \mid \iota \mapsto \llbracket \Gamma, \iota_0:\sigma_0 \vdash P \rrbracket^{\text{game}}) = \llbracket l' \rrbracket (u_0 \mid \iota \mapsto \llbracket \Gamma, \iota_0:\sigma_0 \vdash P \rrbracket^{\text{game}}) \\
&\quad \left[q^{\iota_0} \cdot a^{\iota_0} / \llbracket Q \rrbracket_a (u_0 \mid \iota \mapsto \llbracket \Gamma, \iota_0:\sigma_0 \vdash P \rrbracket^{\text{game}}) \right], \quad q, a \in \llbracket \sigma_0 \rrbracket.
\end{aligned}$$

The two cases are $\iota = l'$ and $\iota \neq l'$. We show the former in detail, the latter is similar. If $\iota = l'$ the evaluation becomes

$$\llbracket \Gamma, \iota_0:\sigma_0 \vdash P \rrbracket^{\text{game}} \left[q^{\iota_0} \cdot a^{\iota_0} / \llbracket Q \rrbracket_a (u_0 \mid \iota \mapsto \llbracket \Gamma, \iota_0:\sigma_0 \vdash P \rrbracket^{\text{game}}) \right].$$

Applying the induction hypothesis on Q this further equals

$$\llbracket \Gamma, \iota_0 : \sigma_0 \vdash P \rrbracket^{\text{game}} \left[q^{\iota_0} \cdot a^{\iota_0} / \llbracket \Gamma \vdash Q [P[M/\iota_0]/\iota M] \rrbracket_a^{\text{game}} \right],$$

for all occurrences of ιM in Q . It is easy to show that in general, for any phrases R, S :

$$\llbracket \Gamma, \iota_0 : \sigma_0 \vdash R \rrbracket^{\text{game}} \left[q^{\iota_0} \cdot a^{\iota_0} / \llbracket \Gamma \vdash S \rrbracket_a^{\text{game}} \right] = \llbracket \Gamma \vdash R[S/\iota_0] \rrbracket^{\text{game}}.$$

Using this property, the evaluation becomes:

$$\llbracket \Gamma \vdash P [Q [P[M/\iota_0]/\iota M] / \iota_0] \rrbracket^{\text{game}},$$

which is immediately seen to be syntactically equal to

$$\llbracket \Gamma \vdash (\iota Q) [P[M/\iota_0]/\iota M] \rrbracket^{\text{game}},$$

for all occurrences of ιM in ιQ , *i.e.* $M = Q$ and all occurrences of ιM in Q .

Finally, for a let phrase $Q = (\mathbf{let} \ \iota'(\iota'_0 : \sigma'_0) = R \ \mathbf{in} \ S)$ we have also several cases. We will show the proof for the following in some detail, the rest are similar:

$$\iota = \iota' \neq \iota'_0$$

$$\begin{aligned} & \llbracket \mathbf{let} \ \iota'(\iota'_0 : \sigma'_0) = R \ \mathbf{in} \ S \rrbracket (u_0 \mid \iota \mapsto \llbracket \Gamma, \iota_0 : \sigma_0 \vdash P \rrbracket^{\text{game}}) \\ &= \llbracket S \rrbracket (u_0 \mid \iota \mapsto \llbracket R \rrbracket (u_0 \mid \iota'_0 \mapsto K_{\sigma'_0}^{\epsilon/\iota'_0} \mid \iota \mapsto \llbracket \Gamma, \iota_0 : \sigma_0 \vdash P \rrbracket^{\text{game}})) \\ &= \llbracket S \rrbracket (u_0 \mid \iota \mapsto \llbracket R \rrbracket (u'_0 \mid \iota \mapsto \llbracket \Gamma, \iota_0 : \sigma_0 \vdash P \rrbracket^{\text{game}})). \end{aligned}$$

In the above, u'_0 is an initial environment for R . By induction hypothesis on R , this further equals:

$$\llbracket S \rrbracket \left(u_0 \mid \iota \mapsto \llbracket \Gamma, \iota'_0 : \sigma'_0 \vdash R [P[M/\iota_0]/\iota M] \rrbracket^{\text{game}} \right),$$

for all ιM occurring in R . By induction on S , this equals:

$$\llbracket \Gamma \vdash S \left[R [P[M/\iota_0]/\iota M] [N/\iota'_0]/\iota N \right] \rrbracket^{\text{game}},$$

for all ιM occurring in R and all ιN in $S[\dots]$. But

$$\begin{aligned} & S \left[R [P[M/\iota_0]/\iota M] [N/\iota'_0]/\iota N \right] \\ &= \mathbf{let} \ \iota'(\iota'_0 : \sigma'_0) = R [P[M/\iota_0]/\iota M] \ \mathbf{in} \ S \quad (\text{by definition of let}) \\ &= (\mathbf{let} \ \iota'(\iota'_0 : \sigma'_0) = R \ \mathbf{in} \ S) [P[M/\iota_0]/\iota M], \end{aligned}$$

since ι occurs bound in R so the substitution takes place only in S . Therefore,

$$\begin{aligned} \llbracket \mathbf{let} \ \iota(\iota'_0:\sigma'_0) = R \ \mathbf{in} \ S \rrbracket (u_0 \mid \iota \mapsto \llbracket \Gamma, \iota_0:\sigma_0 \vdash P \rrbracket^{\text{game}}) \\ = \llbracket (\mathbf{let} \ \iota(\iota'_0:\sigma'_0) = R \ \mathbf{in} \ S) [P[M/\iota_0]/\iota M] \rrbracket^{\text{game}}, \end{aligned}$$

which is what we set out to prove.

The other cases are proved similarly. This concludes the proof. \blacksquare

We proved that the regular language semantics is fully abstract, but relative to the entire IA language. To show that the semantic model is fully abstract for the first-order fragment *qua* stand-alone language we need to prove that it can provide enough contexts to discriminate between all inequivalent phrases.

To prove this we rely on Andrew Pitts's notion of *operational extensionality* (OE) for IA (with active expressions) [21].

Definition 16 (Extensional equivalence) . We say that terms $\Gamma \vdash P : \sigma$ and $\Gamma \vdash Q : \sigma$ are extensionally equivalent, $\Gamma \vdash P \cong_\sigma Q$, if and only if for all substitutions $P[M_i/\iota_i]$ and $Q[M_i/\iota_i]$, such that the results only have free identifiers $\kappa_j : \mathbf{var}\tau$,

$$\begin{aligned} \vdash \mathbf{new}\tau_1 \ \kappa_1 \ \mathbf{in} \ \cdots \ \mathbf{in} \ \mathbf{new}\tau_m \ \kappa_m \ \mathbf{in} \ P[M_i/\iota_i] \\ \equiv_{\text{comm}} \mathbf{new}\tau_1 \ \kappa_1 \ \mathbf{in} \ \cdots \ \mathbf{in} \ \mathbf{new}\tau_m \ \kappa_m \ \mathbf{in} \ Q[M_i/\iota_i]. \end{aligned}$$

Theorem 17 (Operational extensionality)

$$\Gamma \vdash P \equiv_\sigma Q \text{ if and only if } \Gamma \vdash P \cong_\sigma Q.$$

PROOF. Although the definition of extensional equivalence is slightly reformulated, this theorem is essentially a re-stating of Theorem 2.5 in [21]. \blacksquare

The following essential theorem states that contexts for all inequivalences of first-order IA can be found in the language fragment itself, *i.e.* the equivalence relation is well defined with regard to first-order IA:

Theorem 18 (Closed equivalence) For all inequivalent phrases of first-order IA, $\Gamma \vdash P \not\equiv_\theta Q$, there are contexts $\mathcal{C}[-]$ such that $\vdash \mathcal{C}[P] \not\equiv_\sigma \mathcal{C}[Q]$ and $\mathcal{C}[P], \mathcal{C}[Q]$ are closed ground-type terms of first-order IA.

PROOF. For ground types σ we prove this by induction on the number of free identifiers in Γ of type not $\mathbf{var}\tau$.

If Γ consists only of variables identifiers the context is simply the binding of all free variable identifiers to \mathbf{new} . This is the base case.

For the inductive step, assume that for all inequivalences $\Gamma \vdash P \equiv_{\sigma} Q$, witnesses can be found in first order IA. We prove the statement for inequivalences $\Gamma, \iota : \theta \vdash P \equiv_{\sigma} Q$, $\theta \neq \mathbf{var}\tau$:

(1) $\theta = \sigma'$

$\Gamma, \iota : \sigma' \vdash P \not\equiv_{\sigma} Q$ iff, by the OE theorem, $\Gamma, \iota : \sigma' \vdash P \not\equiv_{\sigma} Q$, which, by definition, is the case iff $\Gamma, \Gamma' \vdash P[\iota/M] \not\equiv_{\sigma} Q[\iota/M]$ for some $\Gamma' \vdash M : \sigma'$. But $P[\iota/M] = (\mathbf{let} \ \kappa(\iota : \sigma') = P \ \mathbf{in} \ \kappa M)$, for some fresh identifier κ ; similarly for Q . It follows that

$$\Gamma, \Gamma' \vdash \mathbf{let} \ \kappa(\iota : \sigma') = P \ \mathbf{in} \ \kappa M \not\equiv_{\sigma} \mathbf{let} \ \kappa(\iota : \sigma') = Q \ \mathbf{in} \ \kappa M,$$

which, according to the OE theorem means that

$$\Gamma, \Gamma' \vdash \mathbf{let} \ \kappa(\iota : \sigma') = P \ \mathbf{in} \ \kappa M \not\equiv_{\sigma} \mathbf{let} \ \kappa(\iota : \sigma') = Q \ \mathbf{in} \ \kappa M,$$

where Γ' consists only of variable identifiers. Therefore, environment Γ, Γ' has one less non-variable typed identifier than Γ and we can apply the induction hypothesis, that the inequivalence above also has witnesses in first order IA. But M is in first order IA, being of ground type σ' . Therefore inequivalence $\Gamma, \iota : \sigma' \vdash P \not\equiv_{\sigma} Q$ has a discriminating context in IA.

(2) $\theta = \sigma_1 \times \dots \times \sigma_k \rightarrow \sigma'$.

A similar argument is made, observing that, for $\iota : \theta$:

$$P[\iota/\lambda\iota_1:\sigma_1 \dots \lambda\iota_k:\sigma_k.M] = \mathbf{let} \ \iota(\iota_1:\sigma_1, \dots, \iota_k:\sigma_k) = M \ \mathbf{in} \ P.$$

for $\Gamma, \iota_1:\sigma_1 \dots \lambda\iota_k:\sigma_k \vdash M : \sigma'$. ■

In other words, it is meaningful to speak of equivalence of terms of first-order IA. We can now state our main result:

Theorem 19 (Full abstraction) *The regular language semantics of first order IA is fully abstract:*

$$\Gamma \vdash P \equiv_{\sigma} Q \iff \llbracket P \rrbracket u_0 = \llbracket Q \rrbracket u_0,$$

where initial environment u_0 maps all free identifiers in Γ to copy-cat regular languages.

PROOF. Theorem 18 implies that equivalence remains well defined when restricted to first-order IA and Lemma 15 stipulates that the environment-based semantics is equal to the game semantics. The result of the Full Abstraction Theorem of [2] can be therefore restricted to first-order IA. ■

8 Other syntactic and semantic extensions

In this section we briefly mention some extension, syntactic and semantics, to first-order IA.

The language core defined so far can support arrays as a syntactic extension. Although this addition is not significant from a theoretical point of view, it is interesting from a practical perspective. An imperative first order language with arrays is a basic language for algorithmic programming.

Arrays are introduced as a new ground type of the language: $\sigma ::= \mathbf{array}\tau[n]$, $n \in \mathbb{N}$. The new syntax rules are for array declaration and element access:

$$\frac{\Gamma \vdash V : \mathbf{array}\tau[n] \quad \Gamma \vdash E : \mathbf{expint}}{\Gamma \vdash V[E] : \mathbf{var}\tau} \quad \frac{\Gamma, \iota : \mathbf{array}\tau[m] \vdash C : \mathbf{comm}}{\Gamma \vdash \mathbf{newarray}\tau[m] \ \iota \ \mathbf{in} \ C : \mathbf{comm.}}$$

Semantically, the type and the terms are interpreted as:

$$\begin{aligned} \mathbf{array}\tau[n] &\equiv \underbrace{\mathbf{var}\tau \times \cdots \times \mathbf{var}\tau}_{n \text{ times}} \\ V[E] &\equiv \mathbf{newint} \ \kappa \ \mathbf{in} \ \kappa := E; \mathbf{if} \ \kappa = 1 \ \mathbf{then} \ \pi_1(V) \\ &\quad \mathbf{else} \ \mathbf{if} \ \kappa = 2 \ \mathbf{then} \ \pi_2(V) \\ &\quad \dots \\ &\quad \mathbf{else} \ \mathbf{if} \ \kappa = n \ \mathbf{then} \ \pi_n(V) \ \mathbf{else} \ \mathbf{diverge} \\ \mathbf{newarray}\tau[m] \ \iota \ \mathbf{in} \ C &\equiv \mathbf{new} \ \iota_1 \ \mathbf{in} \ \mathbf{new} \ \iota_2 \ \mathbf{in} \ \dots \ \mathbf{new} \ \iota_m \ \mathbf{in} \ C[(\iota_1, \dots, \iota_m)/\iota] \end{aligned}$$

where π_k is the k -th projection, and identifiers κ , ι_k are fresh.

Deriving the semantic representation from the definition is a simple exercise.

Using arrays, simple data pointers can be also encoded, as indices in a global array; allocation is represented by incrementing a global counter. A pointer is to a cell which contains data and another pointer. These low-level data pointers have been the object of recent research [20]. Since pointers are only indices, pointer arithmetic can be also supported. This in fact is almost identical with how C and C++ represent pointers.

The new type is **ptr**, and the new operations are

$$\frac{\Gamma \vdash P : \mathbf{ptr}}{\Gamma \vdash !P : \mathbf{ptr} \times \mathbf{varint}} \quad \frac{\Gamma \vdash P : \mathbf{ptr} \quad \Gamma \vdash Q : \mathbf{ptr}}{\Gamma \vdash P + Q : \mathbf{ptr}} \quad \frac{}{\Gamma \vdash \mathbf{alloc} : \mathbf{ptr}}$$

We will use a global array called **memory** of size **memsize** along with a global variable **last** recording the last allocated pointer. In memory, every odd-index cell holds another pointer and every even-index cell holds an integer. The

encoding is:

```

ptr  $\equiv$  expint
  ! $P$   $\equiv$  newint  $\iota$  in  $\iota := P$ ; if  $0 < \iota \leq \mathbf{last}$ 
    then (memory $[\iota * 2]$ , memory $[\iota * 2 + 1]$ ) else diverge
alloc  $\equiv$  if last < memsize then last := !last + 2; !last else diverge

```

where ι is a fresh identifier. Garbage collection or deallocation are not modeled.

One feature that is worth mentioning is *tail recursion*. A function is tail-recursive if the recursive call is the last operation in the body of the function. Tail-recursive functions are usually implemented by a syntactic transformation of the function into a while-loop. So, at least in principle, tail-recursive functions can be added to our language; but the details of this transformation are too intricate and depart substantially from the main point of this article. The interested reader can refer to the rich literature on the subject ([4] is one of the pioneering works).

The features mentioned so far are syntactic extensions; they represent simply a direct mapping of the features into already existing ones. But the final extension we will mention is semantic: bounded nondeterminism. The only addition to the language is a random-number generating expression, interpreted as:

$$\llbracket \mathbf{random} \rrbracket_{\mathbf{expint}} = \sum_{n \in \mathcal{N}} q \cdot n.$$

The game semantics of the full IA language with nondeterminism is studied in [11]. We can incorporate bounded non-determinism in first-order IA without requiring further changes to the semantic model.

A more substantial modification to the language is the use of call-by-value function call. A regular-language semantics for such an imperative programming language has been described in [6].

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. The insightful comments of the anonymous referees for the conference paper that preceded this article [7] have greatly contributed to improving the presentation.

References

- [1] S. Abramsky and G. McCusker. Game semantics, lecture notes. <http://www.dcs.ed.ac.uk/home/samson/mdorf97.ps.gz>.
- [2] 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 [18].
- [3] 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.
- [4] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *JACM*, 24(1):44–67, 1977.
- [5] D. R. Ghica. A regular-language model for Hoare-style correctness statements. In *The Second International Workshop on Verification and Computational Logic*, Florence, Italy, Sept. 2001.
- [6] D. R. Ghica. Regular language semantics for a call-by-value programming language. In *Mathematical Foundations of Programming Semantics*, BRICS Notes Series NS-01-2, pages 85–99, Århus, Denmark, May 2001.
- [7] D. R. Ghica and G. McCusker. Reasoning about idealized ALGOL using regular languages. In *Proceedings of 27th International Colloquium on Automata, Languages and Programming ICALP 2000*, volume 1853 of *LNCS*, pages 103–116. Springer-Verlag, 2000.
- [8] C. Hankin and P. Malacaria. Generalised flowcharts and games. *Lecture Notes in Computer Science*, 1443, 1998.
- [9] C. Hankin and P. Malacaria. A new approach to control flow analysis. *Lecture Notes in Computer Science*, 1383, 1998.
- [10] D. Harel. *First-order dynamic logic*, volume 68 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1979. Rev. version of the author’s thesis, M.I.T., 1978.
- [11] R. Harmer and G. McCusker. A fully abstract game semantics for finite nondeterminism. In *14th Symposium on Logic in Computer Science (LICS’99)*, pages 422–430, Washington - Brussels - Tokyo, July 1999. IEEE.
- [12] G. McCusker. *Games and Full Abstraction for a Functional Metalanguage with Recursive Types*. Distinguished Dissertations. Springer-Verlag Limited, 1998.
- [13] 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 [18].

- [14] 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 [18].
- [15] 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>).
- [16] P. W. O’Hearn and J. C. Reynolds. From Algol to polymorphic linear lambda-calculus. *Journal of the Association for Computing Machinery*, 47(1):167–223, Jan. 2000.
- [17] 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 [18].
- [18] P. W. O’Hearn and R. D. Tennent, editors. *ALGOL-like Languages*. Progress in Theoretical Computer Science. Birkhäuser, Boston, 1997. Two volumes.
- [19] F. J. Oles. Functor categories and store shapes. In O’Hearn and Tennent [18], chapter 11, pages 3–12.
- [20] H. Y. Peter O’Hearn, John Reynolds. Local reasoning about programs that alter data structures. In *Proceedings of CSL’01, Paris*, volume 2142 of *LNCS*, pages 1–19. Springer-Verlag, 2001.
- [21] 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 [18].
- [22] 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 [18].
- [23] 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.
- [24] J. C. Reynolds. *The Craft of Programming*. Prentice-Hall International, London, 1981.
- [25] 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 [18].

- [26] J. C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.
- [27] D. A. Schmidt. On the need for a popular formal semantics. *ACM SIGPLAN Notices*, 32(1):115–116, Jan. 1997.
- [28] 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.
- [29] 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, Kôšice, Slovakia, Aug. 1994. Springer-Verlag, Berlin. A version also published as Chapter 15 of [18].