

A Regular-Language Model for Hoare-Style Correctness Statements

Dan R. Ghica¹

*Department of Computing and Information Science,
Queen's University, Kingston,
Ontario, Canada K7L 3N6, +1 (613) 533-6050*

Abstract

Recent developments in game semantics have led to fully abstract models for a variety of theoretically important programming languages. While the full-blown game models tend to be complex and unwieldy for practical purposes, language subsets without higher-order and recursive functions allow greatly simplified models. Two previous papers of the author show how imperative languages with first order functions, local variables and arrays can be modeled using regular languages only. This paper examines the possibility of using these theoretical developments as a basis for model-checking Hoare-style partial correctness statements for such languages. The model-checking problem is shown to be decidable.

1 Introduction

Finding fully abstract denotational models, *i.e.* models that are both correct and complete, for prototypical programming languages, such as PCF, ALGOL or ML, has been an open problem for decades. Only in the last decade fundamental developments in game semantics led to a solution to this problem. A flurry of remarkable theoretical results showed that game-based techniques can give fully abstract models for pure functional languages [2,14,16], imperative languages with local variables and higher-order procedures [3,4], languages with general references [1], languages with generalized control [15] and more.

From a practical perspective, the game models have two attractive features. First, they are fully formal. The moves can be represented by elementary, uninterpreted, symbols; the rules are combinatorial in nature. This means that game models need not deal with the deep semantic issues of set or domain theory. They are, at least on the face of it, suitable for automation. Second, game models are fully abstract, which means that they can be used as a basis

¹ Email: ghica@cs.queensu.ca

for equational reasoning for terms. Equivalences are proved correctly and witnesses can be generated for inequivalences. But game models also have a practical drawback, they are very complex and seemingly unwieldy for usage, beyond proving meta-theoretical results.

This second issue is addressed by two previous papers of the author [10,9]. They show how an imperative language with local variables, arrays and first-order procedures, using call-by-name or call-by-value, can be modeled using regular languages only. The regular-language model is derived directly from the game model using the observation that for this simplified, yet non-trivial, language much of the games apparatus is not necessary. Terms can be modeled by sets of complete plays instead of strategies, sets shown to be regular. The cited papers also show how the regular language model can be used as a reasoning tool to validate term equivalences in the language. The formalism is appealing because of its conceptual simplicity. Also, since the term equivalence problem is formulated as regular language equivalence, it follows that it is decidable in Section 4.

Equational reasoning is theoretically interesting, as it can be used to validate key semantic intuitions, but is not a very useful tool in programming practice. The established formalism for proving correctness of imperative programs is Hoare-style correctness statements. Providing a solid theoretical foundation for a model checking approach to correctness statements is a most interesting application of the regular-language model of imperative languages.

The next section introduces the programming language we will study. Section 3 presents the regular-language model for an imperative first order language with local variables and arrays as well as expressions with side-effects. Having established our programming language model we formulate the model-checking problem for correctness statements and we show it is decidable.

The main purpose of this article is to introduce a new framework for model-checking correctness statements, based on game semantics of programming languages. Establishing the decidability of the model-checking problem is an important first step, but numerous theoretical and practical questions need to be answered. The conclusion of the article mentions some of the most important and outlines a plan for future research.

1.1 Contribution

The language fragment presented here is so common in practice that one may not realize that the interaction of its features (procedures, local variables, arrays, expressions with side-effects) is very difficult to model semantically. Modeling correctness statements in the presence of procedures and local variables has been the subject of extensive research [18, Part V]. Correctness statements for a procedure-less language with expressions with side-effects have been studied in [5]. But all previous research proposes *inference systems* which attempt to solve the verification problem *deductively*. The main

drawback of these approaches is their complexity, caused by the well known *frame problem* ([6] is a recent survey). We suggest a new and much simplified approach: *correctness statements can be verified semantically instead of being inferred deductively*.

It is too early to compare this approach with the state of the art in software model checking such as SLAM [17] or Bandera [7], but we can point out several theoretical advantages. As far as the author is aware, no previous attempts have been made to model-check pre-condition-post-condition-style correctness statements. This is probably because without a fully abstract semantic model it is impossible to validate *program fragments* (as opposed to full programs) as they may include non-locally defined identifiers. Being able to verify program fragments is required to reduce the complexity problem and reason compositionally. The model is decidable (unlike SLAM) and uniform for programs and assertions (unlike Bandera), which bodes well for the future implementation.

2 The programming language

In this paper we will examine a programming language similar to the ones presented in [10,9]. It consists of the simple imperative language, the language of assignment, branching and iteration, to which a non-recursive first-order procedure mechanism is added, along with local (block) variables and local arrays. This is a non-trivial imperative language core which can be used for many algorithmic programming tasks. The language is also technically interesting because the interaction between local variables and procedures has been notoriously difficult to model [18]. The abstract syntax and the typing rules for the language are presented in detail in Appendix A. In the rest of the section we will briefly discuss some of the highlights of this language.

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 (**void**), boolean and integer variables (**varbool**, **varint**) arrays of size m (**arrayint**[m], **arraybool**[m]), and expressions (**expint**, **expbool**), as well as first order functions. The usual arithmetic and logic operators are employed.

The imperative constructs are the usual: 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 presented only for commands, to keep the presentation simple, but within the same framework we could introduce 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.

It is interesting to note that we allow sequencing not only of commands, but also of commands with an expression, resulting in what is called an *active expression*. 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 current imperative languages, such as C.

One notable command of the language is **diverge**, a command that causes a program to enter an unresponsive state similar to that caused by an infinite loop. This command is needed especially for equational reasoning purposes, to cause externally observable behaviour in certain states of the program. The command that performs no operation, similar to the empty command in C or PASCAL, is **skip**.

First order functions are imposed uniformly over the first-order types, including variable and array types. This means that we allow variable and array-returning functions. Function calls can be used as l-values, as in C++. Functions can contain local function definitions, as in PASCAL, arbitrarily nested and subject to the usual scoping rules. The function-call mechanism is call-by-name, for two reasons. First, for a first order imperative language call-by-name allows more expressive power. Commands can be passed to procedures and are not evaluated at the point of call but in the procedure body when the arguments are used. In a call-by-value environment this programming technique requires “wrapping” the command in a function. The second reason is technical, regular-language models for call-by-name being more intuitive. It is important to emphasize that the choice is strictly presentational; regular-language models for similar call-by-value languages also exist [9].

Local variables (**newint**, **newbool**) and arrays (**newarrayint**[m] and **newarraybool**[m]) obey a strict stack discipline. In fact, the call-by-name nature of the language and its stack discipline make it quite similar to idealized ALGOL [20], from which it differs by having active expressions and a restricted procedure mechanism.

3 The regular-language model

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

Definition 3.1 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: $\emptyset, \epsilon \in \mathcal{R}_{\mathcal{A}}$; if $a \in \mathcal{A}$, then $a \in \mathcal{R}_{\mathcal{A}}$;

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

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

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

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

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 $R \in \mathcal{R}_{\mathcal{A}}$, $\tilde{R} \in \mathcal{R}_{\mathcal{A} \cup \overline{\mathcal{A}}}$,

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

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. The broadening of a regular language over alphabet \mathcal{A} is obtained by introducing between consecutive symbols all strings from the complement of the language.

It is a routine exercise to show that:

Proposition 3.2 *Every extended regular expression $R \in \mathcal{R}_{\mathcal{A}}$ denotes a regular language.*

Interpretation is given to terms (sub-programs) of the form $\pi \vdash P:\theta$. This notation means that P is a well formed phrase of type θ , where the type of its free identifiers is given by the type assignment $\pi = \{\iota_1:\theta_1, \dots, \iota_n:\theta_n\}$, metavariable ι ranging over the set of identifiers. The interpretation of P is given by the evaluation function $\llbracket - \rrbracket_{\theta} : [\theta] \rightarrow \llbracket \pi \rrbracket \rightarrow \llbracket \theta \rrbracket$, where $[\theta]$ is the set of all $P:\theta$'s. The set $\llbracket \theta \rrbracket$ includes all regular languages associated with terms in $[\theta]$; the set $\llbracket \pi \rrbracket$ is defined as $\prod_i \llbracket \theta_i \rrbracket$. An element u of this set is called an *environment*. The interpretation of identifiers is given by the environment: $\llbracket \iota \rrbracket_{\theta} u = u(\iota)$. The full interpretation of the language is given in Appendix B.

Every regular language which 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, but the actual computation is given by the sub-expressions into which the meaning of a term can be decomposed. For instance, the meaning of a natural expression E is decomposed in sub-expressions of the form R_E^n , bracketed by opening question q and final answer n . Each describes the plays in E that yield result n for the expression. Similarly, a boolean expression B has two sub-expressions R_B^{tt} and R_B^{ff} corresponding to computations which produce

true and false, respectively. A variable V has two sets of sub-expressions, R_V^n for computations *reading* n from V , and S_V^n for computations *writing* n to V .

It is possible to attach direct computational intuitions to the semantics. For example, **skip** is interpreted by the bracketing moves for commands only, which means that it is a command which completes without having any effects. The regular expression interpreting any arithmetic-logic operator (**op**) is decomposed into p -producing plays, where every such play is any concatenation of plays producing m and n in the arguments, 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. It may be helpful to illustrate the model with a simple equivalence: **while true do** $C \equiv$ **diverge**, where C is any command. Applying the definitions, the meaning of the left-hand side is:

$$\begin{aligned} \llbracket \mathbf{while\ true\ do\ } C \rrbracket u &= run \cdot (R_{\mathbf{true}}^{tt} \cdot R_C)^* \cdot R_{\mathbf{true}}^{ff} \cdot done \\ &= run \cdot (\epsilon \cdot R_C)^* \cdot \emptyset \cdot done = \emptyset = \llbracket \mathbf{diverge} \rrbracket u, \\ \therefore \llbracket \mathbf{true} \rrbracket u &= q \cdot tt = q \cdot \epsilon \cdot tt + q \cdot \emptyset \cdot ff, \quad R_{\mathbf{true}}^{tt} = \epsilon, \quad R_{\mathbf{true}}^{ff} = \emptyset. \end{aligned}$$

In the above example, **diverge** has been interpreted as the empty set of plays. Other than infinite loops, also interpreted by empty sets of plays are terms with numeric *over-flow* and *under-flow*. Approximating the behaviour of arithmetical exceptions as **diverge** is quite crude, but expedient and not unacceptable (see for example [21, Sections 2.7 and 5.1] for a discussion).

Function definition (**let**) is modeled by associating regular expressions called *copy-cat* with the formal arguments. The name has game-semantic origins, because, in copy-cat plays, P repeats every move of O. In application, the (consecutive) moves tagged by the name of the formal argument are replaced by the regular expression denoting the argument (Appendix B.2).

The semantics of the local-variable block rely on the concept of *good variable*, which is simply the property of a variable behaving according to the expected causal rules of assignment and dereferencing: the last value assigned to the variable is always the next value retrieved upon its dereferencing. In an imperative language with procedures variables are not, in general, guaranteed to be good.

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. Both operations are motivated by the way local variables behave. In the block in which it is defined, the variable can be guaranteed to be a good variable. Upon exiting the block all references to it are removed, as all the interactions of the local variable are not “seen” outside its scope. Intersection of regular languages is used to impose the first condition and restriction the second (Appendix B.3). The interpretation of arrays is very similar to that of local variables. An array of size m is the cartesian product of m variables. Good-variable behaviour and restriction are modeled component-

wise (Appendix B.4).

To be able to reason equationally about sub-programs with free identifiers:

$$\iota_1 : \theta_1, \dots, \iota_n : \theta_n \vdash P \equiv_{\theta} Q,$$

we also need to provide interpretations for universal quantifiers over ground and function types (Appendix B.5). In equations, all free identifiers are implicitly universally quantified over their type.

Lemma 3.3 (Representation) *The regular language semantics is a correct representation for the game semantic model of the language fragment.*

Proof. Given the game semantic model of our language, as presented in [3], we observe that for all ground and first order types justification pointers can be reconstructed given any complete play. A routine case analysis shows that all semantic constructs presented here can be identified with the game semantic constructs. \square

From the lemma and the full abstraction theorem in [3], it follows that:

Theorem 3.4 (Full abstraction) *Any two subprograms P and Q in the first order imperative language with local variables and arrays are equivalent if and only if they denote equal regular languages:*

$$\iota_1:\theta_1, \dots, \iota_n:\theta_n \vdash P \equiv_{\theta} Q \iff \llbracket \forall \iota_1:\theta_1 \dots \iota_n:\theta_n. P \rrbracket_{\theta} u_{\emptyset} = \llbracket \forall \iota_1:\theta_1 \dots \iota_n:\theta_n. Q \rrbracket_{\theta} u_{\emptyset}.$$

In the above, u_{\emptyset} is a dummy (empty) environment, all identifiers free in P and Q being bound by universal quantifiers. From the full abstraction theorem and the representation lemma it follows directly that, since equality of regular languages is decidable:

Corollary 3.5 (Decidability) *Equivalence of subprograms of the first order imperative language with local variables and arrays is decidable.*

Decidability of program equivalence may suggest that the regular language model could be used to model-check programs against specifications. However, the state space of an entire program, even reduced with the usual abstraction techniques employed in model checking, is too large for this approach to be feasible. What is needed is a more modular approach, a technique to break a program into pieces that can be model-checked separately. Hoare-style correctness statements are one such technique.

4 Interpreting correctness statements

Assertions are logical formulas expressing properties of “*the state.*” They are much like boolean expressions, but they are different in that they include universal and existential quantifiers. Correctness statements (or specifications,

or Hoare triples), are formulas expressing properties of *commands*. The new typing rules are:

$$\begin{array}{c}
\frac{P : \mathbf{boolexp}}{P : \mathbf{assert}} \quad \frac{\begin{array}{c} [\iota : \theta] \\ \vdots \\ P : \mathbf{assert} \end{array}}{\forall \iota : \theta. P : \mathbf{assert}} \quad \frac{\begin{array}{c} [\iota : \theta] \\ \vdots \\ P : \mathbf{assert} \end{array}}{\exists \iota : \theta. P : \mathbf{assert}} \\
\frac{P : \mathbf{assert} \quad Q : \mathbf{assert}}{P \mathbf{ and } Q : \mathbf{assert}} \quad \frac{P : \mathbf{assert} \quad Q : \mathbf{assert}}{P \mathbf{ or } Q : \mathbf{assert}} \\
\frac{P : \mathbf{assert} \quad C : \mathbf{void} \quad Q : \mathbf{assert}}{P\{C\}Q : \mathbf{spec}}
\end{array}$$

The interpretation of the universal quantifiers is the same as before (Appendix B). Existential quantifiers are interpreted indirectly, using the fact that

$$\exists \iota : \theta. P \equiv \mathbf{not} \forall \iota : \theta. \mathbf{not} P.$$

Hoare triples are usually interpreted like this: if assertion P is true in the initial state, and command C executes successfully, then assertion Q is true in the final state. But the regular-language interpretation of our language is stateless, so this interpretation is not applicable directly. We have to take a slightly indirect approach, and consider what the Hoare triple amounts to *computationally*. Its interpretation is the following equation:

$$\begin{array}{l}
\mathbf{if } P \mathbf{ then } (C; \\
\quad \mathbf{if } Q \mathbf{ then skip} \\
\quad \mathbf{else diverge}) \\
\mathbf{else skip}
\end{array}
\equiv
\begin{array}{l}
\mathbf{if } P \mathbf{ then } (C; \\
\quad \mathbf{if } Q \mathbf{ then skip} \\
\quad \mathbf{else skip}) \\
\mathbf{else skip}
\end{array}$$

In words, if assertion P is true, and command C is executed successfully, assertion Q is true and **diverge** is never executed. Consequently, we can replace it in the left-hand side of the equation with **skip** without changing the overall effect of the program, hence the equivalence. This informal argument can be made rigorous by using a traditional, stateful, semantics of specifications, such as in [13]. Interpreting this equivalence in the regular language model we have that:

$$\begin{aligned}
R_P^{ff} + R_P^{tt} \cdot R_C \cdot (R_Q^{tt} + R_Q^{ff} \cdot \emptyset) &= R_P^{ff} + R_P^{tt} \cdot R_C \cdot (R_Q^{tt} + R_Q^{ff} \cdot \epsilon) \\
\Leftrightarrow R_P^{tt} \cdot R_C \cdot R_Q^{ff} &= \emptyset
\end{aligned}$$

The necessary and sufficient condition for the equivalence to hold is what we will use as the interpretation of correctness statements:

The correctness statement $P\{C\}Q$ holds if and only if the regular language $R_P^{tt} \cdot R_C \cdot R_Q^{ff}$ is empty.

However, correctness statements in this form are not very useful from a practical point of view. The reason is that in order to be able to use an identifier denoting a variable in program C in assertions P and Q , that identifier must occur freely. But a variable which is a free identifier is not guaranteed to be a good variable, so very little can be said about it. For example, the correctness statement $\mathbf{true}\{x := 1\}!x = 1$ does not hold. Moreover, since the expressions we use in assertions may themselves have side-effects, by assigning to variables, not even specifications such as $\mathbf{true}\{\mathbf{skip}\}!x = !x$ are valid.

This problem is addressed by introducing a new type of correctness statement which will be called *universal specifications*, a term which we borrow from Reynolds [19]. Our universal specifications are syntactically and semantically different from Reynolds's, but they have a similar motivation: dealing with properties of non-local objects denoted in the assertions by free identifiers. The universal specifications we use have the following form:

$$\gamma_1(\iota_1), \dots, \gamma_m(\iota_m) \Rightarrow P\{C\}Q,$$

where $\gamma_k(\iota_k)$ represents the assumption that free identifier ι_k has property γ_k . We will talk more about these properties shortly. To interpret universal specifications we need to make the following assumption, which must be met by any concrete property of free identifiers to be defined subsequently:

Assumption 4.1 (Regularity) *For any property γ , its interpretation $\llbracket\gamma(\iota)\rrbracket$ is a regular language.*

Universal specifications are interpreted the obvious way: the comma as a conjunction and the arrow as an implication:

Universal specification $\gamma_1(\iota_1), \dots, \gamma_m(\iota_m) \Rightarrow P\{C\}Q$ is true if and only if for all i , $i \leq m$, and strings w , if $w \in \llbracket\gamma_i(\iota_i)\rrbracket$ then $w \notin (R_P^{tt} \cdot R_C \cdot R_Q^{ff})$.

Two obviously necessary properties are good-variable assumptions for local variables ($\mathbf{gv}_{\iota:\tau}$) and arrays ($\mathbf{ga}_{\iota:\tau[m]}$). Their semantic interpretation, which conforms to the regularity assumption, is given by:

$$\begin{aligned} \llbracket\mathbf{gv}_{\iota:\tau}(\iota)\rrbracket &= \widetilde{\gamma}_\tau^\iota \\ \llbracket\mathbf{ga}_{\iota:\tau[m]}(\iota)\rrbracket &= \bigcap_{0 \leq j \leq m} \widetilde{\gamma}_\tau^{\iota[j]}, \end{aligned}$$

where the regular languages γ_τ^ι and $\gamma_\tau^{\iota[j]}$, are as defined in appendices B.3 and B.4. Variables constrained by a good-variable assumption can be used meaningfully in assertions. Using good-variable assumptions, the previously mentioned putative universal specifications hold, when written as:

$$\begin{aligned} \mathbf{gv}_{x:\mathbf{int}}(x) &\Rightarrow \mathbf{true}\{x := 1\}!x = 1 \\ \mathbf{gv}_{x:\mathbf{int}}(x) &\Rightarrow \mathbf{true}\{\mathbf{skip}\}!x = !x. \end{aligned}$$

The concept of good-variable used here is stronger than the usual one [19], imposing a relation between the *read* and *write* behaviour of the variable

throughout the execution of the command, not only with regard to one assignment only. In traditional terminology, we should call the variable “good-and-not-interfered-with.”

Theorem 4.2 (Model-checking decidability) *The model-checking problem for universal specifications for the first order imperative language with local variables and arrays is decidable (subject to the regularity assumption).*

Proof. The interpretation of correctness statement is equivalent to

$$\bigcap_{i \leq n} \llbracket \gamma_i(\iota_i) \rrbracket \cap (R_P^{\#} \cdot R_C \cdot R_Q^{\#}) = \emptyset.$$

Emptiness is a decidable property of regular languages. □

5 Further research

The overriding intent of this paper is to establish a framework in which we can address the issue of model-checking Hoare-style triples, generalized to universal specifications. The framework proposed in this paper is the game-semantic model of programming languages, simplified to a regular-language model. Hoare triples, and universal specifications, can be verified using a decidable property of regular languages (emptiness). The decidability of the model-checking problem suggests that indeed this framework may serve as a foundation for a practical solution. Another reassuring aspect of our approach is its simplicity and uniformity.

But several important questions need to be researched yet.

From a practical perspective, the next step is to create simple inference-like rules that would permit a degree of compositional reasoning. For example, rules for sequential composition and procedural specifications would permit the model-checking of *parts* of a program separately then establishing properties of the *whole* without repeating the model-checking process. This is needed both to reduce the complexity of the model-checking problem and to support modular software development. Also of practical importance is to incorporate well established data-abstraction techniques [8, and more] in order to reduce the state space. Following that, a proof-of-concept implementation of a model-checker should be possible. This line of research is currently being pursued by the author. In the long term, an efficient algorithmic treatment of the regular language operations required by the model should be an interesting goal.

References

- [1] Abramsky, S., K. Honda and G. McCusker, *A fully abstract game semantics for general references*, in: *Proceedings, Thirteenth Annual IEEE Symposium on Logic in Computer Science*, 1998.
- [2] Abramsky, S., P. Malacaria and R. Jagadeesan, *Full abstraction for PCF*, *Lecture Notes in Computer Science* **789** (1994), pp. 1–59.
- [3] Abramsky, S. 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*, *Electronic notes in Theoretical Computer Science* **3** (1996), also as Chapter 20 of [18].
- [4] Abramsky, S. and G. McCusker, *Full abstraction for Idealized Algol with passive expressions*, *Theoretical Computer Science* **227** (1999), pp. 3–42.
- [5] Boehm, H., *A logic for expressions with side effects*, in: *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, ACM (1982), pp. 268–280.
- [6] Borgida, A., J. Mylopoulos and R. Reiter, *On the frame problem in procedure specifications*, *IEEE Transactions on Software Engineering* **21** (1995), pp. 785–798.
- [7] Corbett, J. and *et al.*, *Bandera: Extracting finite-state models from Java source code*, in: *Proceedings of the 2000 International Conference on Software Engineering*, 2000.
- [8] Cousot, P. and R. Cousot, *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, in: *Fourth ACM Symposium on Principles of Programming Language* (1977), pp. 238–252.
- [9] Ghica, D. R., *Regular language semantics for a call-by-value programming language*, accepted for publication, *Mathematical Foundations of Programming Semantics 17*, Aarhus, Denmark, May 2001.
- [10] Ghica, D. R. and G. McCusker, *Reasoning about idealized ALGOL using regular languages*, in: *Proceedings of 27th International Colloquium on Automata, Languages and Programming ICALP 2000*, LNCS **1853** (2000), pp. 103–116.
- [11] Harel, D., “First-order dynamic logic,” *Lecture Notes in Computer Science* **68**, Springer-Verlag Inc., New York, NY, USA, 1979, 133 pp., rev. version of the author’s thesis, M.I.T., 1978.
- [12] Harel, D., A. R. Meyer and V. R. Pratt, *Computability and completeness in logics of programs*, in: *ACM Symposium on Theory of Computing (STOC ’77)* (1977), pp. 261–268.
- [13] Hoare, C. A. R. and P. E. Lauer, *Consistent and complementary formal theories of the semantics of programming languages*, *Acta Informatica* **3** (1974), pp. 135–153.

- [14] Hyland, J. M. E. and C.-H. L. Ong, *On full abstraction for PCF: I, II and III*, Information and Computation **163** (2000).
- [15] Laird, J., *Full abstraction for functional languages with control*, in: *Proceedings, Twelfth Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press, Warsaw, Poland, 1997, pp. 58–67.
- [16] McCusker, G., “Games and Full Abstraction for a Functional Metalanguage with Recursive Types,” Ph.D. thesis, Department of Computing, Imperial College, University of London (1996).
- [17] Microsoft Corporation, *The SLAM project*.
URL <http://www.research.microsoft.com/projects/slam/>
- [18] O’Hearn, P. W. and R. D. Tennent, editors, “ALGOL-like Languages,” Progress in Theoretical Computer Science, Birkhäuser, Boston, 1997, two volumes.
- [19] Reynolds, J. C., “The Craft of Programming,” Prentice-Hall International, London, 1981.
- [20] Reynolds, J. C., *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 (1981), pp. 345–372, reprinted as Chapter 3 of [18].
- [21] Reynolds, J. C., “Theories of Programming Languages,” Cambridge University Press, 1998.

A The language fragment

Language types:

$$\begin{aligned}
\tau &::= \mathbf{int} | \mathbf{bool} && \text{Data types} \\
\sigma &::= \mathbf{void} | \mathbf{exp}\tau | \mathbf{var}\tau | \mathbf{array}\tau[m], m \in \mathbb{N} && \text{Ground phrase types} \\
\theta &::= \sigma | \sigma_1 \times \sigma_2 \times \dots \times \sigma_k \rightarrow \sigma, k \in \mathbb{N} && \text{Function types,}
\end{aligned}$$

Abstract syntax of terms, with typing judgments:

$$\begin{array}{c}
\frac{}{\mathbf{skip} : \mathbf{void}} \qquad \frac{}{\mathbf{diverge} : \mathbf{void}} \\
\frac{}{\mathbf{true} : \mathbf{expbool}} \qquad \frac{}{\mathbf{false} : \mathbf{expbool}} \\
\frac{}{\mathbf{n} : \mathbf{expint}} \\
\frac{E_1 : \mathbf{expint} \quad E_2 : \mathbf{expint}}{E_1 + E_2 : \mathbf{expint}} \qquad \frac{E_1 : \mathbf{expint} \quad E_2 : \mathbf{expint}}{E_1 - E_2 : \mathbf{expint}} \\
\frac{E_1 : \mathbf{expint} \quad E_2 : \mathbf{expint}}{E_1 * E_2 : \mathbf{expint}} \qquad \frac{E_1 : \mathbf{expint} \quad E_2 : \mathbf{expint}}{E_1 / E_2 : \mathbf{expint}} \\
\frac{E_1 : \mathbf{expint} \quad E_2 : \mathbf{expint}}{E_1 = E_2 : \mathbf{expbool}} \qquad \frac{E_1 : \mathbf{expint} \quad E_2 : \mathbf{expint}}{E_1 < E_2 : \mathbf{expbool}} \\
\frac{B_1 : \mathbf{expbool} \quad B_2 : \mathbf{expbool}}{B_1 \text{ and } B_2 : \mathbf{expbool}} \qquad \frac{B_1 : \mathbf{expbool} \quad B_2 : \mathbf{expbool}}{B_1 \text{ or } B_2 : \mathbf{expbool}} \\
\frac{C_1 : \mathbf{void} \quad C_2 : \mathbf{void}}{C_1; C_2 : \mathbf{void}} \qquad \frac{C : \mathbf{void} \quad E : \mathbf{exp}\tau}{C; E : \mathbf{exp}\tau} \\
\frac{B : \mathbf{expbool} \quad C : \mathbf{void}}{\mathbf{while } B \text{ do } C : \mathbf{void}} \qquad \frac{B : \mathbf{bool} \quad C_1 : \mathbf{void} \quad C_2 : \mathbf{void}}{\mathbf{if } B \text{ then } C_1 \text{ else } C_2} \\
\frac{V : \mathbf{var}\tau \quad E : \mathbf{exp}\tau}{V := E : \mathbf{void}} \qquad \frac{V : \mathbf{var}\tau}{!V : \mathbf{exp}\tau} \\
\frac{[\iota : \sigma_0 \times \dots \times \sigma_m \rightarrow \sigma'] \quad [\iota_0 : \sigma_0 \dots \iota_m : \sigma_m]}{\frac{Q : \sigma \qquad P : \sigma'}{\mathbf{let } \iota(\iota_0 : \sigma_0, \dots, \iota_m : \sigma_m) = P \text{ in } Q : \sigma}} \\
\frac{\iota : \sigma_0 \times \dots \times \sigma_m \rightarrow \sigma \quad Q_0 : \sigma_0 \quad \dots \quad Q_m : \sigma_m}{\iota(Q_0, \dots, Q_m) : \sigma}
\end{array}$$

$$\begin{array}{c}
[\iota : \mathbf{var}\tau] \\
\vdots \\
C : \mathbf{void} \\
\hline
\mathbf{new}\tau \iota \text{ in } C : \mathbf{void}
\end{array}
\qquad
\begin{array}{c}
[\iota : \mathbf{array}\tau[m]] \\
\vdots \\
C : \mathbf{void} \\
\hline
\mathbf{newarray}\tau[m] \iota \text{ in } C : \mathbf{void}
\end{array}$$

$$\frac{\iota : \mathbf{array}\tau[m] \quad E : \mathbf{expint}}{\iota[E] : \mathbf{var}\tau}$$

In the above, metavariable ι ranges over the set of valid identifiers, $m \in \mathbb{N}$.

B The language model

With every data-type τ and phrase-type θ we associate an alphabet $\mathcal{A}[\tau]$:

$$\begin{aligned}
\mathcal{A}[\mathbf{int}] &= \mathcal{N} = \{n \mid -N < n < N\}, & \mathcal{A}[\mathbf{bool}] &= \{tt, ff\} \\
\mathcal{A}[\mathbf{comm}] &= \{run, done\}, \\
\mathcal{A}[\mathbf{exp}\tau] &= \{q, v \mid v \in \mathcal{A}[\tau]\}, \\
\mathcal{A}[\mathbf{var}\tau] &= \{read, v, write(v), ok \mid v \in \mathcal{A}[\tau]\}, \\
\mathcal{A}[\mathbf{array}\tau[m]] &= \{read[j], v[j], write[j](v), ok[j] \mid 0 \leq j < m, v \in \mathcal{A}[\tau]\}, \\
\mathcal{A}[\sigma_1 \times \sigma_2 \times \dots \times \sigma_k \rightarrow \sigma] &= \{a_i \mid a \in \mathcal{A}[\sigma_k], 1 \leq i \leq k\} \cup \mathcal{A}[\sigma].
\end{aligned}$$

B.1 Basic terms

$$\begin{aligned}
\llbracket \mathbf{skip} \rrbracket_{\mathbf{void}} u &= run \cdot done, & \llbracket \mathbf{diverge} \rrbracket_{\mathbf{void}} u &= \emptyset, & \llbracket n \rrbracket_{\mathbf{expint}} u &= q \cdot n, \\
\llbracket \mathbf{true} \rrbracket_{\mathbf{expbool}} u &= q \cdot tt, & \llbracket \mathbf{false} \rrbracket_{\mathbf{expbool}} u &= q \cdot ff, \\
\llbracket \mathbf{not } B \rrbracket &= q \cdot R_B^{tt} \cdot ff + q \cdot R_B^{ff} \cdot tt, \\
\llbracket E_1 \mathbf{op} E_2 \rrbracket_{\mathbf{exp}\tau} u &= \sum_{m \mathbf{op} n=p} q \cdot R_{E_1}^m \cdot R_{E_2}^n \cdot p \\
\llbracket C; C' \rrbracket_{\mathbf{void}} u &= run \cdot R_C \cdot R_{C'} \cdot done, \\
\llbracket C; E \rrbracket_{\mathbf{exp}\tau} u &= \sum_{v \in \mathcal{A}[\tau]} q \cdot R_C \cdot R_E^v \cdot v, \\
\llbracket \mathbf{while } B \mathbf{do} C \rrbracket_{\mathbf{void}} u &= run \cdot (R_B^{tt} \cdot R_C)^* \cdot R_B^{ff} \cdot done, \\
\llbracket \mathbf{if } B \mathbf{then } C \mathbf{else } C' \rrbracket_{\mathbf{void}} u &= run \cdot (R_B^{tt} \cdot R_C + R_B^{ff} \cdot R_{C'}) \cdot done \\
\llbracket V := M \rrbracket_{\mathbf{void}} u &= \sum_v run \cdot R_M^v \cdot S_V^v \cdot done \\
\llbracket !V \rrbracket_{\mathbf{var}\tau} u &= \sum_v q \cdot R_V^v \cdot v,
\end{aligned}$$

where the R^-, S^- regular expressions are defined using these decompositions:

$$\begin{aligned}
\llbracket E \rrbracket_{\text{expint}} u &= \sum_{n \in \mathcal{N}} q \cdot R_E^n \cdot n, \\
\llbracket B \rrbracket_{\text{expbool}} u &= q \cdot R_B^{tt} \cdot tt + q \cdot R_B^{ff} \cdot ff, \\
\llbracket C \rrbracket_{\text{void}} u &= \text{run} \cdot R_C \cdot \text{done}, \\
\llbracket V \rrbracket_{\text{var}\tau} u &= \sum_{n \in \mathcal{A}[\tau]} (\text{read} \cdot R_V^n \cdot n) + \sum_{n \in \mathcal{A}[\tau]} (\text{write}(n) \cdot S_V^n \cdot \text{ok}).
\end{aligned}$$

B.2 Application and abstraction

$$\llbracket \iota(Q_1, \dots, Q_m) \rrbracket_{\sigma} u = u(\iota)[a_i^t \cdot b_i^t / R_{Q_i}], \text{ where } \llbracket Q_i \rrbracket_{\sigma_i} u = \sum_{a, b \in \mathcal{A}[\sigma_i]} a \cdot R_{Q_i} \cdot b,$$

$$\begin{aligned}
\llbracket \text{let } \iota(\iota_0 : \sigma_0, \dots, \iota_m : \sigma_m) = P \text{ in } Q \rrbracket_{\sigma} u \\
= \llbracket Q \rrbracket_{\sigma} (u | \iota \mapsto \llbracket P \rrbracket_{\sigma'} (u | \iota_0 \mapsto R_{\sigma_0}^{\iota_0} | \dots | \iota_m \mapsto R_{\sigma_m}^{\iota_m})).
\end{aligned}$$

The regular expressions $R_{\sigma_i}^{\iota_i}$ are:

$$\begin{aligned}
R_{\text{void}}^{\iota_i} &= \text{run} \cdot \text{run}_i^{\iota_i} \cdot \text{done}_i^{\iota_i} \cdot \text{done}, \\
R_{\text{exp}\tau}^{\iota_i} &= \sum_{n \in \mathcal{A}[\tau]} q \cdot q_i^{\iota_i} \cdot n_i^{\iota_i} \cdot n \\
R_{\text{var}\tau}^{\iota_i} &= \sum_{n \in \mathcal{A}[\tau]} \text{read} \cdot \text{read}_i^{\iota_i} \cdot n_i^{\iota_i} \cdot n + \sum_{n \in \mathcal{A}[\tau]} \text{write}(n) \cdot \text{write}(n)_i^{\iota_i} \cdot \text{ok}_i^{\iota_i} \cdot \text{ok}, \\
R_{\text{array}\tau[m]}^{\iota_i} &= \sum_{n \in \mathcal{A}[\tau]} \text{read} \cdot \text{read}[j]_i^{\iota_i} \cdot n[j]_i^{\iota_i} \cdot n + \sum_{n \in \mathcal{A}[\tau]} \text{write}(n) \cdot \text{write}[j](n)_i^{\iota_i} \cdot \text{ok}[j]_i^{\iota_i} \cdot \text{ok}.
\end{aligned}$$

B.3 Local variables

$$\llbracket \text{new}\tau \iota \text{ in } M \rrbracket_{\text{void}} u = (\widetilde{\gamma}_{\tau}^{\iota} \cap \llbracket M \rrbracket_{\text{void}} (u | \iota \mapsto R_{\text{var}\tau}^{\iota})) |_{\mathcal{A}_{\tau}^{\iota}}, \text{ where}$$

$$\begin{aligned}
\gamma_{\tau}^{\iota} &= (\sum_{v \in \mathcal{A}[\tau]} \text{write}(v)^{\iota} \cdot \text{ok}^{\iota} \cdot (\text{read}^{\iota} \cdot v^{\iota})^*)^*, \\
R_{\text{var}\tau}^{\iota} &= \sum_{n \in \mathcal{A}[\tau]} \text{read} \cdot \text{read}^{\iota} \cdot n^{\iota} \cdot n + \sum_{n \in \mathcal{A}[\tau]} \text{write}(n) \cdot \text{write}(n)^{\iota} \cdot \text{ok}^{\iota} \cdot \text{ok}, \\
\mathcal{A}_{\tau}^{\iota} &= \{\text{write}(v)^{\iota}, \text{ok}^{\iota}, \text{read}^{\iota}, v^{\iota} | v \in [\tau]\}.
\end{aligned}$$

B.4 Arrays

$$\begin{aligned}
\llbracket \iota[E] \rrbracket_{\text{var}\tau} u &= \sum_{m \in \mathcal{N}} \sum_{n \in \mathcal{A}[\tau]} \text{read} \cdot R_E^m \cdot \text{read}[m]^{\iota} \cdot n[m]^{\iota} \cdot n \\
&\quad + \sum_{m \in \mathcal{N}} \sum_{n \in \mathcal{A}[\tau]} \text{write}(n) \cdot R_E^m \cdot \text{write}[m](n)^{\iota} \cdot \text{ok}[m]^{\iota} \cdot \text{ok}, \\
\llbracket \text{newarray}\tau[m] \iota \text{ in } M \rrbracket u &= \bigcap_{0 \leq j < m} \widetilde{\gamma}_{\tau}^{\iota[j]} \cap \llbracket M \rrbracket \left(u | \iota \mapsto \sum_{0 \leq j < m} R_{\tau}^{\iota[j]} \right) \Big|_{\bigcup_{0 \leq j < m} \mathcal{A}_{\tau}^{\iota[j]}},
\end{aligned}$$

where

$$\begin{aligned}
R_\tau^{\iota[j]} &= \sum_{n \in \mathcal{A}[\tau]} \text{read}[j] \cdot \text{read}[j]^\iota \cdot n[j]^\iota \cdot n[j] + \sum_{n \in \mathcal{A}[\tau]} \text{write}[j](n) \cdot \text{write}[j](n)^\iota \cdot \text{ok}[j]^\iota \cdot \text{ok}[j], \\
\gamma_\tau^{\iota[j]} &= \left(\sum_{v \in \mathcal{A}[\tau]} \text{write}[j](v) \cdot \text{ok}[j]^\iota \cdot (\text{read}[j]^\iota \cdot v[j]^\iota)^* \right)^*, \\
\mathcal{A}^{\iota[j]}_\tau &= \{ \text{write}(v)^{\iota[j]}, \text{ok}^{\iota[j]}, \text{read}^{\iota[j]}, v^{\iota[j]} \mid v \in \llbracket \tau \rrbracket \}.
\end{aligned}$$

B.5 Quantifiers

$$\llbracket \forall \iota : \theta . P \rrbracket u = \llbracket P \rrbracket (u \mid \iota \mapsto I_\theta^\iota)$$

where

$$I_{\text{void}}^\iota = \text{run} \cdot \text{run}^\iota \cdot \text{done}^\iota \cdot \text{done}$$

$$I_{\text{exp}\tau}^\iota = \sum_{n \in \mathcal{A}[\tau]} q \cdot q^\iota \cdot n^\iota \cdot n$$

$$I_{\text{var}\tau}^\iota = \sum_{n \in \mathcal{A}[\tau]} \text{read} \cdot \text{read}^\iota \cdot n^\iota \cdot n + \sum_{n \in \mathcal{A}[\tau]} \text{write}(n) \cdot \text{write}(n)^\iota \cdot \text{ok}^\iota \cdot \text{ok}$$

$$\begin{aligned}
I_{\text{array}\tau[m]}^\iota &= \sum_{0 \leq j < m} \sum_{n \in \mathcal{A}[\tau]} \text{read}[j] \cdot \text{read}[j]^\iota \cdot n[j]^\iota \cdot n[j] \\
&\quad + \sum_{0 \leq j < m} \sum_{n \in \mathcal{A}[\tau]} \text{write}[j](n) \cdot \text{write}[j](n)^\iota \cdot \text{ok}[j]^\iota \cdot \text{ok}[j]
\end{aligned}$$

$$I_{\sigma_1 \times \dots \times \sigma_n \rightarrow \text{void}}^\iota = \text{run} \cdot \text{run}^\iota \cdot (R_{\sigma_1 \times \dots \times \sigma_n}^\iota)^* \cdot \text{done}^\iota \cdot \text{done}$$

$$I_{\sigma_1 \times \dots \times \sigma_n \rightarrow \text{exp}\tau}^\iota = \sum_{n \in \mathcal{A}[\tau]} q \cdot q^\iota \cdot (R_{\sigma_1 \times \dots \times \sigma_n}^\iota)^* \cdot n^\iota \cdot n$$

$$\begin{aligned}
I_{\sigma_1 \times \dots \times \sigma_n \rightarrow \text{var}\tau}^\iota &= \sum_{n \in \mathcal{A}[\tau]} \text{read} \cdot \text{read}^\iota \cdot (R_{\sigma_1 \times \dots \times \sigma_n}^\iota)^* \cdot n^\iota \cdot n \\
&\quad + \sum_{n \in \mathcal{A}[\tau]} \text{write}(n) \cdot \text{write}(n)^\iota \cdot (R_{\sigma_1 \times \dots \times \sigma_n}^\iota)^* \cdot \text{ok}^\iota \cdot \text{ok}
\end{aligned}$$

$$\begin{aligned}
I_{\sigma_1 \times \dots \times \sigma_n \rightarrow \text{array}\tau[m]}^\iota &= \sum_{0 \leq j < m} \sum_{n \in \mathcal{A}[\tau]} \text{read}[j] \cdot \text{read}[j]^\iota \cdot (R_{\sigma_1 \times \dots \times \sigma_n}^\iota)^* \cdot n[j]^\iota \cdot n[j] \\
&\quad + \sum_{0 \leq j < m} \sum_{n \in \mathcal{A}[\tau]} \text{write}[j](n) \cdot \text{write}[j](n)^\iota \cdot (R_{\sigma_1 \times \dots \times \sigma_n}^\iota)^* \cdot \text{ok}[j]^\iota \cdot \text{ok}[j]
\end{aligned}$$

$$R_{\sigma_1 \times \dots \times \sigma_n}^\iota = \sum_{1 \leq i \leq n} R_{\sigma_i}^{\iota, i}$$