Electronic Notes in Theoretical Computer Science 45 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume45.html 13 pages

Regular-Language Semantics for a Call-by-Value Programming Language

Dan R. Ghica [1,2]

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

Abstract

We explain how game semantics can be used to reason about term equivalence in a finitary imperative first order language with arrays. For this language, the game-semantic interpretation of types and terms is fully characterized by their sets of complete plays. Because these sets are regular over the alphabet of moves, they are representable by (extended) regular expressions. The formal apparatus of game semantics is greatly simplified but the good theoretical properties of the model are preserved. The principal advantage of this approach is that it is mathematically elementary, while fully formalized. Since language equivalence for regular languages is decidable, this method of proving term equivalence is suitable for automation.

1 Introduction

In the last decade the use of game semantics in the analysis of programming languages has yielded numerous remarkable theoretical results. Most importantly, this innovative approach led to fully abstract models for languages that have been under semantic scrutiny for decades, such as PCF [10,2] and idealized Algol [4,6]. The theoretical success of game semantics is well complemented by an elegantly articulated and intuitive conceptual association between key language features (such as mutable state or control) and neat combinatorial constraints on strategies (such as *innocence* or *bracketing*)[3].

There is, however, a frustrating aspect of game semantics. While the models proposed are fully abstract, which means that in principle they correctly validate all program equivalences and inequivalences, they are at the same time so intricate that applying them to that end is often a Gordian task. What is needed is an adequate notation that would allow a *calculus* of games,

[1] This author acknowledges the support of a NSERC PGSB grant.

[2] Email: ghica@cs.queensu.ca

to make proofs less prolix and more formal. While a wieldy and accessible calculus that captures the full power of games may be unattainable, calculi for restricted yet non-trivial subsets of game-based models are very handy. They illustrate the game semantics in an applied setting, making the subject more accessible to those who find its abstractness daunting. But, more importantly, such calculi can actually serve as a foundation for a new and practical approach to program analysis, predicated on solid theoretical results. A similar avenue of research, but as applied to static analysis, is explored by Hankin and Malacaria [8,9].

In a previous paper [7] we showed how a greatly simplified games model of idealized Algol can be used to give elementary proofs to semantically relevant putative equivalences that have been an important part of the study of the language [12]. This paper follows a similar approach, but focuses on a different language, an imperative call-by-value first order language with arrays. This language is important from a practical point of view; it is the idiom in which many common programs, for example for searching or sorting, are written. For this language we present what we believe to be a practicable semantic calculus which can be used for validating term (subprogram) equivalences. Since equivalent subprograms can be replaced in any context, without restrictions, the technique presented here can be applied to both program development through refinement and to program maintenance. We are optimistic about the practical application of such a technique because it is mathematically elementary and calculational, both features considered essential requirements for a "popular semantics" [15]. Moreover, this calculus is fully formal. Because it is based on regular expressions, language equivalence is decidable, which makes it suitable for automation.

2 Foil: a first order imperative language

In this paper we are concerned with a simple, prototypical, programming language that can be found at the core of most of today's imperative languages. It combines the features of the simple imperative language (mutable local variables, control structures) with a recursion-free first-order procedure mechanism based on the simply typed call-by-value lambda calculus, and an elementary data structure facility (arrays). The data sets of Foil are finite, as is the case with realistic programming languages. The decision to set aside higher order procedures and recursion is dictated by the need to confine the formalism to regular expressions only; they are not expressive enough to represent these more powerful features. Another restriction is to allow only uncurried functions, but this is only for the sake of simplicity of presentation. Curried functions can be readily added and explicated within the bounds of the same formalism.

Foil has three kinds of types: return types, argument types and function types. The return types are the most elementary, and can be returned by functions. They are the "values" of the language: booleans, naturals and a type of commands, **void**, similar to that in C or Java.

$$\tau ::= \mathbf{bool} \mid \mathbf{nat} \mid \mathbf{void}.$$

The argument types include the return types plus all the other types that can be passed as arguments to functions, which are variables, arrays and tuples:

$$\sigma ::= \tau \mid \mathbf{var} \mid \mathbf{array}[\mathbf{n}] \mid \sigma \times \sigma, \quad \mathbf{n} \in \mathbb{N}.$$

The types above and function types, from argument to return types, form the type system of the language:

$$\theta ::= \sigma \mid \sigma \to \tau.$$

The terms of the language are associated with typing judgments of the form:

$$\iota_1 \colon \theta_1, \iota_2 \colon \theta_2, \ldots, \iota_k \colon \theta_k \vdash M \colon \theta,$$

where ι_i are free variables.

The terms of the language are constants, operators, free variables, control structures, command composition, variable declaration, array declaration, array element selection, assignment, dereferencing and function declaration and application (Figure 1).

3 Game semantics of Foil

The reader is not expected to be familiar with game semantics in order to understand this article. Also, it is not possible to condense such a rich topic in a few pages, but good and comprehensive introductory material is available [3]. The specific games model used to interpret Foil and on which we base the present regular language model is the one developed by Abramsky and McCusker [5,1]. In this section we will only introduce some of the key intuitive concepts of game semantics, especially as applied to call-by-value games.

The concept of *game* employed by game semantics is a broad one: "an activity conducted according to prescribed rules." Computation is represented as a dialogical game between two protagonists: *Player* (P) represents the program and *Opponent* (O) represents the environment, or the context, in which the program is run. The interaction between O and P consists of a sequence of moves, governed by rules. For example, O and P need to take turns and every move needs to be *justified* by a preceding move. The moves are of two kinds, *questions* and *answers*; one of the rules constraining the interplay is that every answer must correspond to the last unanswered question (*bracketing*).

To every type in the language corresponds a *game*; that is, the set of all possible sequences of moves, together with the way in which they are

$$n\colon \mathbf{nat} \qquad\qquad \mathbf{true}\colon \mathbf{bool} \qquad\qquad \mathbf{false}\colon \mathbf{bool}$$

$$\mathbf{skip}\colon \mathbf{void} \qquad \mathbf{diverge}\colon \theta$$

$$\frac{M\colon \mathbf{nat} \quad N\colon \mathbf{nat}}{M+N\colon \mathbf{nat}} \qquad \frac{M\colon \mathbf{nat} \quad N\colon \mathbf{nat}}{M=N\colon \mathbf{bool}} \qquad \frac{M\colon \mathbf{bool} \quad N\colon \mathbf{bool}}{M \ \mathbf{and}\ N\colon \mathbf{bool}}$$

$$\frac{M\colon \mathbf{bool} \quad N\colon \mathbf{void} \quad P\colon \mathbf{void}}{\mathbf{if}\ M\ \mathbf{then}\ N\ \mathbf{else}\ P\colon \mathbf{void}} \qquad \frac{M\colon \mathbf{bool} \quad N\colon \mathbf{void}}{\mathbf{while}\ M\ \mathbf{do}\ N\colon \mathbf{void}}$$

$$\frac{M\colon \mathbf{void} \quad N\colon \tau}{M;N\colon \tau} \qquad \frac{M\colon \mathbf{var} \quad N\colon \mathbf{nat}}{M := N\colon \mathbf{void}} \qquad \frac{M\colon \mathbf{var}}{!M\colon \mathbf{nat}}$$

$$\frac{\begin{array}{c}[\iota_1\colon \sigma_1] \quad \cdots \quad [\iota_k\colon \sigma_k]\\ \vdots \\ M\colon \tau\end{array}}{\lambda\iota_1\ldots\iota_k : \sigma_1\ldots\sigma_k.M\colon \sigma_1\times\cdots\times\sigma_k \to \tau}$$

$$\frac{M_1\colon \sigma_1 \quad \cdots \quad M_k\colon \sigma_k}{(M_1,\ldots,M_k)\colon \sigma_1\times\cdots\times\sigma_k} \qquad \frac{\begin{array}{c}[\iota\colon \mathbf{var}]\\ \vdots \\ M\colon \mathbf{void}\end{array}}{\mathbf{new}\ \iota\ \mathbf{in}\ M\colon \mathbf{void}} \qquad \frac{\begin{array}{c}\left[\iota\colon \mathbf{array}[\mathbf{n}]\right]\\ \vdots \\ M\colon \mathbf{void}\end{array}}{\mathbf{new}\ \iota[n]\ \mathbf{in}\ M\colon \mathbf{void}}$$

$$\frac{F\colon \sigma\to\tau \quad M\colon \sigma}{FM\colon \tau} \qquad \frac{\iota\colon \mathbf{array}[\mathbf{n}] \quad N\colon \mathbf{nat}}{\iota[N]\colon \mathbf{var}}$$

Fig. 1. Terms and typing judgments

justified within the sequence. A program is represented as a set of sequences of moves in the appropriate game, more precisely as a *strategy* for that game: a predetermined way for P to respond to O's moves. The semantic models which provide full abstraction for call-by-name languages are developed within this general games framework.

In an influential paper, Moggi showed that call-by-value languages are interpreted in a Cartesian Closed Category (CCC) with coproducts and a *strong monad* [13]. If a CCC has infinite co-products then its free completion under co-product produces the required monadic structure. The games framework forms indeed a CCC, with games as objects and strategies as morphisms, and McCusker showed how co-product games can be added [11]. These two ideas are incorporated in [5] to create a category of games suitable for interpreting call-by-value languages. Arriving at the concrete call-by-value games presented here is only a matter of carrying out in enough detail the categorical construction.

The resulting games are, however, interesting in their own right because they offer some basic insight into call-by-value computation. A type is not represented by a game, but by a *family of games.* A strategy interpreting a term has two distinct stages, a *protocol* stage in which one of the members of the family is selected, followed by a play in the selected game. Intuitively, this mirrors the fact that, in call-by-value, all arguments are evaluated exactly once before the body of the function is evaluated. Accordingly, free identifiers can have only one value throughout the evaluation of a term. In contrast, call-by-name allows identifiers to correspond to different values at various points in the evaluation.

This point is illustrated by the following example: $\mathrm{f}\colon \mathbf{nat} \to \mathbf{nat}, \mathrm{x}\colon \mathbf{nat} \vdash \mathrm{f}(\mathrm{x})\colon \mathbf{nat}$. For call-by-name a typical play is:

f: **nat** → **nat**		x: **nat**	⊢	f(x): **nat**
				q
	q			
q				
		q		
		n		
n				
q				
		q		
		n'		
n'				
	m			
				m

O asks for the value of f(x); P asks for the value returned by f; O asks for the argument of f; P asks for the value of x; O answers n; P relays that answer back to O; O asks again for the argument of f and the same cycle repeats; O answers with m to the value returned by f; P relays that answer back to O, answering the initial question. Notice that in the course of the play the value of x can be requested several times, and since the answer is given by O it may change. Now let us look at the same term evaluated under call-by-value:

f: **nat** → **nat**		x: **nat**	⊢	f(x): **nat**
				$?$
			$?$	
			n	
			$?(n)$	
	$?(n)$			
	m			
				m

The moves under the ⊢ symbol are not part of the play, but they are some of the concealed activities that are part of the protocol. The play is: O asks for the value of f(x); as a result of the protocol, P asks for the value of f *in component* n; O answers m; P relays the answer back, answering the initial question. Only part of the protocol is shown: P asks in what component should the play proceed; O answers with a component index n for x; P requests that the game should continue in component n.

In [5] it was shown that this games framework gives a fully abstract model for call-by-value. Moreover, by relaxing one of the constraints on strategies (*innocence*) the same article shows how a fully abstract model for an imperative language with ML-style data references can be defined, using a *good-variable* non-innocent strategy to model mutable store. These ideas are further expanded in [1] to show that dropping another constraint on strategies (*visibility*) gives rise to a fully abstract model for general references (references to data, procedures, higher order functions, other references).

4 Regular language semantics of types

A game for a type, or a strategy for a term, is fully characterized by its set of plays together with the way moves are justified. But if the language is sufficiently restricted then there is only one way in which moves can be justified within a play sequence—Foil is such a restricted language. This means that such languages can be fully characterized by plays taken to be sequences of moves only. Moreover, the sequences of moves are regular sets, so they can be denoted by (extended) regular expressions. This is an approach we took before, in dealing with Algol [7]. For the restricted language, extended regular expressions give a convenient, compact, fully formal calculus, quite handy in defining the semantics of actual programs. The regular-language semantics arises out of the model in [5], by working out the details of the categorical construction.

Definition 4.1 *The set $\mathcal{R}_{\mathcal{A}}$ of extended regular expressions R over a finite alphabet $\mathcal{A}$ is defined as:*

$$\begin{array}{ll}
R ::= \emptyset \mid \epsilon \mid a, \quad a \in \mathcal{A} & \textit{Constants}, \\
R ::= R \cdot R \mid R + R \mid R \cap R & \textit{Operators}, \\
R ::= R^* & \textit{Iteration}, \\
R ::= R|_{\mathcal{A}'}, \quad \mathcal{A}' \subseteq \mathcal{A} & \textit{Hiding}, \\
R ::= R\langle v \rangle & \textit{Indexing.}
\end{array}$$

Most of the above are standard regular expression constructs, to which we add intersection and two new operations, *hiding* and *indexing.* The latter are operations on regular languages that can be carried out directly at the level of regular expressions. Hiding represents a restriction of a regular expression to a subset of its alphabet by removing all the occurrences of symbols in the restricted alphabet; its language is the set of sequences of the original languages with all the elements of $\mathcal{A}'$ deleted. Indexing is defined as the tagging of the first symbol a of any sequence in the language with the string

$$\llbracket\theta\rrbracket = \left(\textstyle\sum_{c\in C_\theta} P^c_\theta\right)\cdot \textstyle\sum_{k\in K_\theta} R_\theta(k)^*,\ \text{where:}$$

$$\begin{array}{rcl}
\llbracket\tau\rrbracket & : & P^v_\tau = ?\cdot v,\quad R_\tau=\epsilon,\quad K_\tau=\{\star\},\quad C_{\mathbf{void}}=\{\star\}\\
 & & C_{\mathbf{nat}}=\{0,1,\ldots,n_N\}=N,\quad C_{\mathbf{bool}}=\{tt,\mathit{ff}\}\\
\llbracket\sigma_1\times\sigma_2\rrbracket & : & P^c_{\sigma_1\times\sigma_2}=?\cdot c,\quad C_{\sigma_1\times\sigma_2}=C_{\sigma_1}\times C_{\sigma_2}\\
 & & R_{\sigma_1\times\sigma_2}(k)=\begin{cases}R_{\sigma_1}(k) & \text{if } k\in K_{\sigma_1}\\ R_{\sigma_2}(k) & \text{if } k\in K_{\sigma_2}\end{cases}, K_{\sigma_1\times\sigma_2}=K_{\sigma_1}\uplus K_{\sigma_2}\\
\llbracket\mathbf{var}\rrbracket & : & P_{\mathbf{var}}=?\cdot\star,\quad C_{\mathbf{var}}=\{\star\},\quad K_{\mathbf{var}}=N\cup\{\star\}\\
 & & R_{\mathbf{var}}=\sum_{m\in N}\mathit{read}{\cdot}m+\sum_{m\in N}\mathit{write}(m){\cdot}\star\\
\llbracket\mathbf{array}[\mathbf{n}]\rrbracket & : & P_{\mathbf{array}[\mathbf{n}]}=?\cdot\star,\quad C_{\mathbf{array}[\mathbf{n}]}=\{\star\}\\
 & & K_{\mathbf{array}[\mathbf{n}]}=\{i|i<n\}\cup\{i|i<n\}\times N\\
 & & R_{\mathbf{array}[\mathbf{n}]}=\sum_{\substack{m\in N\\ i<n}}\mathit{read}(i){\cdot}m+\sum_{\substack{m\in N\\ i<n}}\mathit{write}(i,m){\cdot}\star\\
\llbracket\sigma\to\tau\rrbracket & : & P_{\sigma\to\tau}=?\cdot\star,\quad C_{\sigma\to\tau}=\{\star\}\\
 & & R_{\sigma\to\tau}=\sum_{c\in K_{\sigma\to\tau}}\big(q(c)\cdot\sum_{d\in C_\tau}R_\sigma\langle c\rangle\cdot d\big)^*,\quad K_{\sigma\to\tau}=C_\sigma
\end{array}$$

Fig. 2. Semantics of Foil types

v, resulting in $a(uv)$, where u is the pre-existing tag of R, possibly empty (ϵ).

A regular-language representation of the game semantics of Foil is defined as follows. With types we associate games, represented as regular languages over an alphabet denoting the moves. With terms we associate strategies, represented as regular languages over the disjoint sum of the alphabets of the types of the free identifiers and the term itself.

As mentioned in the previous sections, a call-by-value game for a type θ has two stages: a protocol-game P^c_θ followed by a component-game $R_\theta = \sum_{k\in K_\theta} R_\theta(k)$. The protocol part of the play corresponds to the co-product structure which forms the monad, tying together the various components.

A play in the protocol game always has the form $?\cdot c$ for $c \in C_\theta$. We call C_θ *the set of component-selecting moves.* The first move in a play in the component game R_θ has the form $m(k)$, where $k \in K_\theta$. We call K_θ *the set of component-defining moves*; every such component is represented by the regular language $R_\theta(k)$. Notice that C_θ and K_θ are distinct sets. If sets C or K only have one element ($\star$), we will often omit it as an index.

The regular language semantics of Foil types is the one given in Figure 2. For **void**, **nat** and **bool** the definition is straightforward. Variables **var** are the product of an acceptor and an expression type, not reified in the actual language. Arrays of size n are identified with products of n variables. Proving these regular expressions correctly represent games is tedious, but routine.

In the case of product and function types it is required that the alphabets (sets of moves) of the types involved are disjoint. This is achieved by systematically tagging all the moves in each alphabet with tags uniquely associated with each type occurrence.

5 Regular language semantics of terms

Terms in Foil are interpreted as *families of regular expressions*, representing the call-by-value strategies. They have the following form, where P and R are the protocol and the component parts:

$$[\![\iota_1 : \theta_1, \ldots, \iota_n : \theta_n \vdash M : \theta]\!] = \biguplus_{c \in \prod_{i \leq n} C_{\theta_i}} \left\{ \textstyle\sum_{k \in K_\theta} P_M^{c,k} \cdot R_M(k)^* \right\}.$$

Free identifiers are interpreted as:

$$[\![\iota : \theta \vdash \iota : \theta]\!] \quad : \quad P_{\iota:\,\theta}^{c,k} = P_\theta^c, \quad k \in K_\theta, \quad R_\iota = R_\theta[m/m{\cdot}m^\iota][n/n^\iota{\cdot}n],$$

for all moves m of odd index in the play (the O-moves) and n of even index (P-moves). Since in the regular expressions moves always occur in pairs, this substitution can be carried out directly on the regular expression defining the plays. This "doubling-up" of moves is the representation of the important *copy-cat* strategy of game semantics. The new tag ι is meant to differentiate between the two occurrences of type θ, in the environment and in the term itself. For example:

$$\begin{aligned} [\![\mathrm{x} : \mathbf{nat} \vdash \mathrm{x} : \mathbf{nat}]\!] &= \{? \cdot n \cdot \epsilon \mid n \in N\}, \\ [\![\mathrm{f} : \mathbf{nat} \to \mathbf{nat} \vdash \mathrm{f} : \mathbf{nat} \to \mathbf{nat}]\!] &= \{?{\cdot}\star{\cdot}R_f^*\} \\ &= \left\{?{\cdot}\star{\cdot}\left(\textstyle\sum_{i,n \in N} q(i){\cdot}q(i)^f{\cdot}n^f{\cdot}n\right)^*\right\}. \end{aligned}$$

For all basic constants of the language we have $R = \epsilon$ and:

$$\begin{array}{lll} P_{n:\,\mathbf{nat}} = ? \cdot n, & P_{\mathbf{true}:\,\mathbf{bool}} = ? \cdot tt, & P_{\mathbf{diverge}:\,\theta} = \emptyset, \\ P_{\mathbf{skip}:\,\mathbf{void}} = ? \cdot \star, & P_{\mathbf{false}:\,\mathbf{bool}} = ? \cdot f\!f. & \end{array}$$

Binary arithmetic, logic and arithmetic-logic operators can be interpreted as abbreviations involving predefined functions, for which the semantics of application (to be defined later) will be used to compose the meanings of subphrases:

$$[\![+ : \mathbf{nat} \times \mathbf{nat} \to \mathbf{nat}]\!] \quad : \quad P_+^{m,n} = ? \cdot \star, \quad R_+(m,n) = ?(m,n){\cdot}(m+n),$$

with $m, n \in N$. Similarly for all other operators. Sequencing is:

$$\begin{aligned}
[\![-;-\colon \mathbf{void}\times\mathbf{void}\to\mathbf{void}]\!] \quad &: \quad P_{;} = ?\cdot\star, \quad R_{;}=\epsilon,\\
[\![-;-\colon \mathbf{void}\times\mathbf{nat}\to\mathbf{nat}]\!] \quad &: \quad P_{;} = \sum_{n\in N} ?\cdot n, \quad R_{;}=\epsilon.
\end{aligned}$$

Assignment and dereferencing are respectively:

$$\begin{aligned}
[\![-{:=}-\colon \mathbf{var}\times\mathbf{nat}\to\mathbf{void}]\!] \quad &: \quad P^{n}_{:=} = ?\cdot\star, \quad R_{:=}(n) = \mathit{write}(n)\cdot\star,\\
[\![!-\colon \mathbf{var}\to\mathbf{nat}]\!] \quad &: \quad P_{!} = ?\cdot\star, \quad R_{!} = \sum_{n\in N} \mathit{read}\cdot n.
\end{aligned}$$

Abstraction is defined as explicitly reindexing the regular expressions denoting the meaning of a term M with the component moves of the types of identifiers abstracted over:

$$\begin{aligned}
&[\![\Gamma\vdash \lambda\iota_1\ldots\iota_k : \sigma_1\times\cdots\times\sigma_k.M\colon \sigma_1\times\cdots\times\sigma_k\to\tau]\!] :\\
&\qquad P^{k}_{\lambda} = ?\cdot\star,\ R_{\lambda}(k) = Q\langle k\rangle,\ k\in\prod_{i\leq n} C_{\sigma_i},\ Q\in [\![\Gamma,\iota_1\colon\sigma_1,\ldots,\iota_k\colon\sigma_k\vdash M]\!] .
\end{aligned}$$

The most important, and the most complex, is the meaning of application:

$$\begin{aligned}
[\![\Gamma\vdash MN\colon\tau]\!] \quad &: \quad P_{MN} = ?\cdot P^{c'}_{N}\cdot R'_{M}(c)[x\cdot y/R^{x,y}_{N}], \quad R_{MN}=\epsilon\\
\text{where} \quad & \quad ?\cdot P^{c'}_{N}\cdot c\cdot R_N \in [\![\Gamma''\vdash N\colon\sigma]\!] , \quad R_N = \sum_{x,y} x\cdot R^{x,y}_{N}\cdot y\\
& \quad ?\cdot\star\cdot\sum_{c} q(c)\cdot R'_{M}(c)\in [\![\Gamma'\vdash M\colon\sigma\to\tau]\!] , \quad R_\sigma = \sum_{x,y} x\cdot y.
\end{aligned}$$

The regular expressions and regular expression families involved in the definition above are well defined in general, with one exception. If N is a diverging term then either one of $P^{c'}_{N}$ and R_N may be $\emptyset$ with the other arbitrary, $\emptyset$ being a zero-element for concatenation. This ambiguity is resolved by always choosing $P^{c'}_{N}=\emptyset$, to be consistent with the fact that $P_{\mathbf{diverge}}=\emptyset$, as presented earlier. The choice for R_N is then irrelevant, ϵ by convention.

The semantics of application is derived directly from the game semantics as well, more precisely from compositions of strategies. In composing strategies, which is how application is interpreted, the moves in the game (type occurrence) through which composition is realized serve as "triggers" which switch the thread of execution between the two strategies. In our particular case, whenever such a move x occurs, a regular expression denoting the trace of execution for the argument is inserted in the regular expression denoting the body of the function, up to the point where another context-switching move y occurs. In the process of composing strategies all trigger moves are subsequently hidden. Here, the key technique is to decompose a regular expression into smaller regular expressions and, using systematic substitution and concatenation, create the regular expressions corresponding to the result. This technique will be also used in defining the regular language semantics of terms which are not abbreviations.

Since functions are not curried we need to define pairing. It reflects the left-to-right order of argument evaluation in function call, specific to Foil:

$$
\begin{aligned}
&[\![\Gamma \vdash (M,N) \colon \sigma \times \sigma']\!] :\\
&P_{M,N}^{(c,c')(k,k')} = ?\cdot Q_M^{c,k}\cdot Q_N^{c',k'}\cdot(c,c'), \quad R_{M,N}(k,k') = \big(R_M(k) + R_N(k')\big)^*\\
&\text{ where } P_M^{c,k} = ?\cdot Q_M^{c,k}\cdot c, \quad P_N^{c,k'} = ?\cdot Q_N^{c',k'}\cdot c'.
\end{aligned}
$$

Branching and looping are defined directly, not as abbreviations:

$$
\begin{aligned}
&[\![\textbf{if } B \textbf{ then } M \textbf{ else } N]\!] &:&\quad P_{\textbf{if}} = (?\cdot P_B^{tt}\cdot P_M'\cdot\star) + (?\cdot P_B^{f\!f}\cdot P_N'\cdot\star), \quad R_{\textbf{if}} = \epsilon,\\
&[\![\textbf{while } B \textbf{ do } M]\!] &:&\quad P_{\textbf{while}} = ?\cdot(P_B^{tt}\cdot P_M')^*\cdot P_B^{f\!f}\cdot\star, \quad R_{\textbf{while}} = \epsilon,\\
&\text{where} &:&\quad P_B = \sum_{v\in\{tt,f\!f\}} ?\cdot P_B^v\cdot v, \quad P_M = ?\cdot P_M'\cdot\star, \quad P_N = ?\cdot P_N'\cdot\star\ .
\end{aligned}
$$

The semantics of **if** is directly specified in the games semantics. Looping in game semantics is defined as an abbreviation using the recursion combinator. A general recursion combinator is not specified in this treatment, but the fixed point can be calculated by hand; the semantics of **while** above is the result of that calculation.

Array element access is also directly defined:

$$
\begin{aligned}
&[\![\Gamma \vdash \iota[N] : \textbf{var}]\!] \quad :\\
&P_{\iota[N]}^k = ?\cdot P_N^{k'}\cdot\star, \quad R_{\iota[N]}(k) = \sum_{m\in N} \mathit{read}(k)\cdot m + \sum_{m\in N} \mathit{write}(k,m)\cdot\star,\\
&\text{where } P_N^{k'} = ?\cdot P_N^k\cdot k.
\end{aligned}
$$

Finally, as in the case of Algol, local variables are realized by imposing a *good variable* restriction on the plays and by *hiding* the actions of the local variables. Good-variable behaviour simply means that the last value written in a variable will be the next value read from that variable; this restriction is imposed using intersection with the following regular expression, associated with a variable ι:

$$\gamma^\iota = \overline{\mathcal{A}_\iota}^*\cdot\left(\overline{\mathcal{A}_\iota}^*\cdot\textstyle\sum_{n\in N}\mathit{write}(n)^\iota\cdot\star^\iota\cdot\overline{\mathcal{A}_\iota}^*\cdot(\mathit{read}^\iota\cdot n^\iota\cdot\overline{\mathcal{A}_\iota}^*)^*\right)^*,$$

where $\mathcal{A}_\iota = \{\mathit{read}^\iota, \mathit{write}(n)^\iota, n^\iota, \star^\iota | n \in N\}$ is the set of all moves tagged by ι, *i.e.* all moves involving variable ι. Therefore local variable definition is:

$$[\![\textbf{new } \iota \textbf{ in } M]\!] = \{\,(\gamma^\iota \cap Q)\,|_{\mathcal{A}_\iota} \quad | \, Q \in [\![M]\!]\}\,.$$

For similar reasons, the meaning of array declaration is:

$$[\![\textbf{new } \iota[n] \textbf{ in } M]\!] = \left\{\left(\bigcap_{i\le n}\gamma^{\iota[i]} \cap Q\right)\Bigg|_{\bigcup_{i\le n}\mathcal{A}_{\iota[i]}} \quad \Bigg| \, Q \in [\![M]\!]\right\},$$

$$\gamma^{\iota[i]} = \overline{\mathcal{A}_{\iota[i]}}^*\cdot\left(\overline{\mathcal{A}_{\iota[i]}}^*\cdot\textstyle\sum_{n\in N}\mathit{write}(n,i)^\iota\cdot\star^\iota\cdot\overline{\mathcal{A}_{\iota[i]}}^*\cdot(\mathit{read}(i)^\iota\cdot n^\iota\cdot\overline{\mathcal{A}_{\iota[i]}}^*)^*\right)^*, i \le n.$$

This concludes the semantic definition of Foil. We can state that:

Lemma 5.1 (Representation) *The regular language semantics of* Foil *is a fully correct representation of the games and strategies used in the game semantic model.*

From this, it follows directly from [5] that:

Theorem 5.2 (Full Abstraction) *The regular language semantics of* Foil *is fully abstract,* i.e. *two terms of* Foil *are equivalent if and only if they denote the same family of regular languages:*

$$\text{For all } \Gamma \vdash P, Q \colon \theta, \quad P \equiv Q \iff [\![P]\!] = [\![Q]\!].$$

In addition, since the representation is by regular languages, for which language equality is decidable, it follows directly that:

Theorem 5.3 (Decidability) *Equivalence of two terms of* Foil *is decidable.*

6 Example of reasoning

Since one of the stated purposes of this article is to provide a basis for a new and potentially practical tool, we think it is important to show in some detail an example. Space constraints prevent us from presenting a realistic program, so we will instead prove a simple, but theoretically important, equivalence of Foil:

$$\mathrm{f} \colon \mathbf{nat} \to \mathbf{void}, \mathrm{v} \colon \mathbf{nat} \vdash \quad \mathbf{new}\ \mathrm{x}\ \mathbf{in}\ \mathrm{x} := \mathrm{v}; \mathrm{f}(!\mathrm{x}) \equiv_{\mathbf{void}} \mathrm{f}(\mathrm{v}).$$

Proof:

$$[\![\mathrm{x} \colon \mathbf{var} \vdash \mathrm{x} \colon \mathbf{var}]\!] : P_x = ? \cdot \star,$$

$$R_x = \sum_{n \in N} \mathit{read} \cdot \mathit{read}^x \cdot n^x \cdot n + \sum_{m \in N} \mathit{write}(m) \cdot \mathit{write}(m)^x \cdot \star^x \cdot \star$$

$$[\![\mathrm{x} \colon \mathbf{var} \vdash !\mathrm{x} \colon \mathbf{nat}]\!] : P_{!x} = ? \cdot \sum_{n \in N} \cdot \mathit{read}^x \cdot n^x \cdot n, \quad R_{!x} = \epsilon$$

$$[\![\mathrm{f} \colon \mathbf{nat} \to \mathbf{void} \vdash \mathrm{f} \colon \mathbf{nat} \to \mathbf{void}]\!] : P_f = ? \cdot \star, \quad R_f = \left(\textstyle\sum_{i \in N} q(i) \cdot q(i)^f \cdot \star^f \cdot \star\right)^*$$

$$[\![\mathrm{f} \colon \mathbf{nat} \to \mathbf{void}, \mathrm{x} \colon \mathbf{var} \vdash \mathrm{f}(!\mathrm{x}) \colon \mathbf{nat}]\!] :$$

$$P_{f(!x)} = ? \cdot \sum_{n \in N} \mathit{read}^x \cdot n^x \cdot \left(q(n)^f \cdot \star^f\right)^* \cdot \star, \quad R_{f(!x)} = \epsilon$$

$$[\![\mathrm{v} \colon \mathbf{nat}, \mathrm{x} \colon \mathbf{var} \vdash \mathrm{x} := \mathrm{v} \colon \mathbf{void}]\!] : P^v_{x:=v} = ? \cdot \mathit{write}(v)^x \cdot \star^x \cdot \star, \quad R_{x:=v} = \epsilon$$

$$[\![\mathrm{f} \colon \mathbf{nat} \to \mathbf{void}, \mathrm{v} \colon \mathbf{nat}, \mathrm{x} \colon \mathbf{var} \vdash \mathrm{x} := \mathrm{v}; \mathrm{f}(!\mathrm{x}) \colon \mathbf{void}]\!] :$$

$$P^v_{x:=v; f(!x)} = ? \cdot \mathit{write}(v)^x \cdot \star^x \cdot \sum_{n \in N} \mathit{read}^x \cdot n^x \cdot \left(q(n)^f \cdot \star^f\right)^* \cdot \star, \quad R_{x:=v; f(!x)} = \epsilon$$

$$[\![\mathrm{f} \colon \mathbf{nat} \to \mathbf{void}, \mathrm{v} \colon \mathbf{nat} \vdash \mathbf{new}\ \mathrm{x}\ \mathbf{in}\ \mathrm{x} := \mathrm{v}; \mathrm{f}(!\mathrm{x}) \colon \mathbf{void}]\!] :$$

$$P^v = ? \cdot \cancel{\mathit{write}(v)^x \cdot \star^x \cdot \mathit{read}^x \cdot v^x} \cdot \left(q(v)^f \cdot \star^f\right)^* \cdot \star = ? \cdot \left(q(v)^f \cdot \star^f\right)^* \cdot \star, \quad R = \epsilon.$$

Therefore:

$$\begin{aligned}&[\![\mathrm{f}\colon \mathbf{nat} \to \mathbf{void}, \mathrm{v}\colon \mathbf{nat} \vdash \mathbf{new}\ \mathrm{x}\ \mathbf{in}\ \mathrm{x} := \mathrm{v}; \mathrm{f}(!\mathrm{x})\colon \mathbf{void}]\!]\\&\quad = \left\{?\cdot \left(q(v)^f \cdot \star^f\right)^* \cdot \star \,\middle|\, v \in N\right\} = [\![\mathrm{f}\colon \mathbf{nat} \to \mathbf{void}, \mathrm{v}\colon \mathbf{nat} \vdash \mathrm{f}(\mathrm{v})\colon \mathbf{void}]\!]\,.\end{aligned}$$

7 Conclusion

We have presented a games-based regular language semantics for an imperative language with first order procedures using call-by-value function application, with arrays and variables passed by-reference. The model is obtained directly from the game semantic model [5,1] by working out the details of the category-theoretical presentation and by observing that much of the games apparatus (justification pointers, *etc.*) is unnecessary in handling the present language subset.

Acknowledgement

Guy McCusker's suggestions and explanations were essential in the writing of this paper, I owe him a great deal. I would like to thank Bob Tennent for his support and encouragement. Many thanks are due to the anonymous referees for providing insightful comments and pertinent corrections.

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, *Game semantics*, lecture notes, 1997 Marktoberdorf summer school (available from http://www.dcs.ed.ac.uk/home/samson/mdorf97.ps.gz).

[4] Abramsky, S. and G. McCusker, *Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions*, in: O'Hearn and Tennent [14] pp. 297–329, two volumes.

[5] Abramsky, S. and G. McCusker, *Call-by-value games*, in: *CSL: 11th Workshop on Computer Science Logic*, LNCS **1414**, 1998, pp. 1–17.

[6] Abramsky, S. and G. McCusker, *Full abstraction for Idealized Algol with passive expressions*, Theoretical Computer Science **227** (1999), pp. 3–42.

[7] 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.

[8] Hankin, C. and P. Malacaria, *Generalised flowcharts and games*, Lecture Notes in Computer Science **1443** (1998).

[9] Hankin, C. and P. Malacaria, *Non-deterministic games and program analysis: an application to security*, in: *Proceedings, Fourteenth Annual IEEE Symposium on Logic in Computer Science*, 1999 pp. 443–452.

[10] Hyland, J. M. E. and C.-H. L. Ong, *On full abstraction for PCF: I, II and III*, Information and Computation **163** (2000).

[11] McCusker, G., "Games and Full Abstraction for a Functional Metalanguage with Recursive Types," Distinguished Dissertations, Springer-Verlag Limited, 1998.

[12] Meyer, A. R. 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* (1988), pp. 191–203, reprinted as Chapter 7 of [14].

[13] Moggi, E., *Notions of computation and monads*, Information and Computation **93** (1991), pp. 55–92.

[14] O'Hearn, P. W. and R. D. Tennent, editors, "Algol-like Languages," Progress in Theoretical Computer Science, Birkhäuser, Boston, 1997, two volumes.

[15] Schmidt, D. A., *On the need for a popular formal semantics*, ACM SIGPLAN Notices **32** (1997), pp. 115–116.