ALGORITHMIC GAME SEMANTICS

A Tutorial Introduction

SAMSON ABRAMSKY (samson@comlab.ox.ac.uk)
Oxford University Computing Laboratory

1. Introduction

Game Semantics has emerged as a powerful paradigm for giving semantics to a variety of programming languages and logical systems. It has been used to construct the first syntax-independent fully abstract models for a spectrum of programming languages ranging from purely functional languages to languages with non-functional features such as control operators and locally-scoped references [4, 21, 5, 19, 2, 22, 17, 11]. A substantial survey of the state of the art of Game Semantics *circa* 1997 was given in a previous Marktoberdorf volume [6].

Our aim in this tutorial presentation is to give a first indication of how Game Semantics can be developed in a new, algorithmic direction, with a view to applications in computer-assisted verification and program analysis. Some promising steps have already been taken in this direction. Hankin and Malacaria have applied Game Semantics to program analysis, e.g. to certifying secure information flows in programs [25]. A particularly striking development was the work by Ghica and McCusker [15] which captures the game semantics of a fragment of Idealized Algol in a remarkably simple form as regular expressions. This leads to a decision procedure for observation equivalence on this fragment. Ghica has subsequently extended the approach to a call-by-value language with arrays [14], and to model checking Hoare-style program correctness assertions [13].

We believe the time is ripe for a systematic development of this algorithmic approach to game semantics. Game Semantics has several features which make it very promising from this point of view. It provides a very *concrete* way of building *fully abstract* models. It has a clear operational content, while admitting *compositional methods* in the style of denotational semantics. The basic objects studied in Game Semantics are games, and strategies on games. Strategies can be seen as certain kinds of highly-constrained processes, hence they admit the same kind of automata-theoretic representations central to model checking and allied methods in computer-assisted verification. At the same time, games and strategies naturally form themselves into rich mathematical structures which yield very accurate models of advanced high-level programming languages, as the various full abstraction results show. Thus the promise of this approach is to carry over the methods of model checking, which has been so effective in the analysis of circuit designs and communications protocols, to much more *structured* programming situations, in which data-types as well as control flow are important.

A further benefit of the algorithmic approach is that by embodying game semantics in tools, and making it concrete and algorithmic, it should become more accessible and meaningful to practitioners. We see Game Semantics as having the potential to fill the role of a "Popular Formal Semantics" called for in an eloquent paper by David Schmidt [31], which can help to bridge the gap between the semantics and programming language communities. Game Semantics has been successful in its own terms as a semantic theory; we aim to make it useful to and usable by a wider community.

In relation to the extensive current activity in software model checking and computer assisted verification (see e.g. [8, 12]), our approach is distinctive, being founded on a highly-structured *compositional* semantic model. This means that we can directly apply our methods to *program phrases* (i.e. terms-in-context with free variables) in a high-level language with procedures, local variables and data types; moreover, the soundness of our methods is guaranteed by the properties of the semantic models on which they are based. By contrast, most current model checking applies to relatively "flat" unstructured situations, in which the system being analyzed is presented as a transition system or automaton. Our aim is to build on the tools and methods which have been developed in the verification community, while exploring the advantages offered by our semantics-based approach.

1.1. OVERVIEW

In the following section, we shall begin with an informal overview of game semantics, followed by a step-by-step development of how constructs in a procedural programming language can be modelled in this approach. We formalize these descriptions using elementary tools from formal language theory; this will guarantee that the semantic descriptions are themselves effective, and can serve as the basis for model-checking and program analysis.

A more systematic account is then given in section 3, while section 4 discusses model-checking.

2. Game semantics for a procedural language

2.1. INFORMAL INTRODUCTION

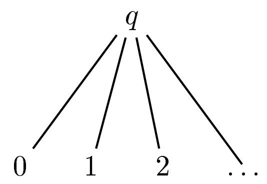
Before proceeding to a precise formalization, we will give an informal presentation of the main ideas through examples, with the aim of conveying how close to programming intuitions the formal model is.

As the name suggests, game semantics models computation as the playing of a certain kind of game, with two participants, called Player (P) and Opponent (O). P is to be thought of as representing the system under consideration, while O represents the environment. In the case of programming languages, the system corresponds to a term (a piece of program text) and the environment to the context in which the term is used. This is a key point at which games models differ from other process models: the distinction between the actions of the system and those of its environment is made explicit from the very beginning. (For a fuller discussion of the ramifications of this distinction, see [1]).

In the games we shall consider, O always moves first—the environment sets the system going—and thereafter the two players make moves alternately. What these moves are, and when they may be played, are determined by the rules of each particular game. Since in a programming language a *type* determines the kind of computation which may take place, types will be modelled as games; a *program* of type A determines how the system behaves, so programs will be represented as *strategies* for P, that is, by the specification of responses by P to the moves O may make.

2.2. MODELLING VALUES

In standard denotational semantics, values are *atomic*: a natural number is represented simply as $n \in \omega$. In game semantics, each number is modelled as a simple interaction: the environment starts the computation with an initial move q (a *question*: "What is the number?"), and P may respond by playing a natural number (an *answer* to the question). So the game $\mathbb{N}$ of natural numbers looks like this:



and the strategy for $\mathsf{3}$ is "When O plays q, I will play 3":

$$\begin{array}{cc} I\!\!N & \\ q & \mathsf{O} \\ 3 & \mathsf{P} \end{array}$$

In diagrams such as the above, time flows downwards: here O has begun by playing q, and at the next step P has responded with 3, as the strategy dictates.

2.3. EXPRESSIONS

The interactions required to model expressions are a little more complex. The view taken in game semantics is that the environment of an expression consumes the output and provides the input, while the expression itself consumes the input and produces the output. Thus the game involved in evaluating an expression such as

$$x : I\!\!N, y : I\!\!N \vdash x + y : I\!\!N$$

is formed from "three copies of $I\!\!N$", one for each of the inputs x and y, and one for the output—the result of evaluating $x + y$. In the output copy, O may demand output by playing the move q and P may provide it. In the input copies, the situation is reversed: P may demand input with the move q. Thus the O/P role of moves in the input copy is reversed. A typical computation of a natural strategy interpreting this expression has the following form.

$$\begin{array}{cccccc} I\!\!N & , & I\!\!N & \vdash & I\!\!N & \\ & & & & q & \mathsf{O} \\ q & & & & & \mathsf{P} \\ 3 & & & & & \mathsf{O} \\ & & q & & & \mathsf{P} \\ & & 2 & & & \mathsf{O} \\ & & & & 5 & \mathsf{P} \end{array}$$

The play above is a particular run of the strategy modelling the addition operation:

> "When O asks for output, I will ask for my first input x; when O provides input m for x, I will ask for my second input y; when O provides the input n for y, I will give output $m + n$."

It is important to notice that the play in each copy of $I\!\!N$ (that is, each column of the above diagram) is indeed a valid play of $I\!\!N$: it is not possible for O to begin with the third move shown above, supplying an input to the function immediately.

This example also illustrates the *intensional* character of game semantics. The above strategy for the addition operation is only one possibility; another would be the strategy which evaluated the two arguments in the opposite order. These would be *distinct* strategies for computing the *same* operation (function). This distinction may appear otiose in the purely functional setting; but the ability to make such distinctions becomes crucial when we come to model non-functional features such as state or control.

2.4. INTERACTION: COMPOSITION OF STRATEGIES

Game semantics is intended to provide a *compositional* interpretation of programs: just as small programs can be put together to form large ones, so strategies can be combined to form new strategies. The fundamental "glue" in traditional denotational semantics is function application; for game semantics it is *interaction* of strategies which gives us a notion of composition.

Consider the example of addition again, with the type

$$x : \mathbb{N}, y : \mathbb{N} \vdash x + y : \mathbb{N}$$

We would like to combine this with $\vdash 3 : \mathbb{N}$:

$$\frac{\vdash 3 : \mathbb{N} \quad x : \mathbb{N}, y : \mathbb{N} \vdash x + y : \mathbb{N}}{y : \mathbb{N} \vdash 3 + y : \mathbb{N}}$$

This is just *substitution* of 3 for x, or in logical terms, the Cut rule [16]. In order to represent this composition, we let the two strategies interact with one another. When `add` plays a move in the first copy of $\mathbb{N}$ (corresponding to asking for the value of x), we feed it as an O-move to the strategy for 3; conversely, when this strategy responds with 3 in $\mathbb{N}$, we feed this move as an O-move back to `add`.

$$\begin{array}{ccccc} \mathbb{N} & , & \mathbb{N} & \vdash & \mathbb{N} \\ &&&& q \\ q &&&& \\ 3 &&&& \\ && q && \\ && 5 && \\ &&&& 8 \end{array}$$

By hiding the action in the first copy of $\mathbb{N}$, we obtain the strategy

$$\begin{array}{ccc} \mathbb{N} & \vdash & \mathbb{N} \\ && q \\ q && \\ n && \\ && n+3 \end{array}$$

representing the (unary) operation which adds 3 to its argument, as expected. So in game semantics, composition of functions is modelled by CSP-style "parallel composition + hiding" [18, 3].

2.5. VARIABLES AND COPY-CAT STRATEGIES

To interpret a variable

$$x : I\!N \vdash x : I\!N$$

we play as follows:

$$\begin{array}{ccc} I\!N & \vdash & I\!N \\ & & q \\ q & & \\ n & & \\ & & n \end{array}$$

This is a basic example of a *copy-cat strategy*. Note that at each stage the strategy simply copies the preceding move by O from one copy of $I\!N$ to the other. This is clearly not specific to $I\!N$—the same idea can be applied to any game. Copy-cat strategies have a fundamental importance in game semantics, as first recognized in [3]. Note that they provide *identities* with respect to composition. For example, if we form the composition

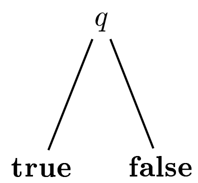
$$\frac{z : I\!N \vdash z : I\!N \qquad x : I\!N, y : I\!N \vdash x + y : I\!N}{z : I\!N, y : I\!N \vdash z + y : I\!N}$$

then the corresponding strategy is *the same strategy for addition*.

By repeatedly applying composition to the strategies for constants, variables, and operations such as addition, we can build up interpretations for arbitrary expressions

$$x_1 : B_1, \ldots, x_k : B_k \vdash e : B$$

where B_1, ..., B_n, B are basic data types such as $I\!N$ and $\mathbf{bool}$, the latter interpreted by the game



An important additional example of an operation is the *conditional*

$$b : \mathbf{bool}, x : B, y : B \vdash \mathtt{cond}(b, x, y) : B$$

which plays as follows

$$
\begin{array}{cccc}
\mathbf{bool} & ,\ B & ,\ B & \vdash\ B \\
 & & & q \\
q & & & \\
\mathbf{true} & & & \\
 & q & & \\
 & a & & \\
 & & & a
\end{array}
\qquad\qquad
\begin{array}{cccc}
\mathbf{bool} & ,\ B & ,\ B & \vdash\ B \\
 & & & q \\
q & & & \\
\mathbf{false} & & & \\
 & & q & \\
 & & b & \\
 & & & b
\end{array}
$$

The strategy for conditional is:

> "Ask for the boolean argument; if I get $\mathbf{true}$, I will play copy-cat between the second argument and the result; if I get $\mathbf{false}$, I will play copy-cat between the third argument and the result".

2.6. FORMALIZATION

Before we continue with the development of the semantics, we will set up a simple framework in which we can make the definitions formal and precise. We will interpret each type of the programming language by an *alphabet* of "moves". A "play" of the game will then be interpreted as a *word* (string, sequence) over this alphabet. A strategy will be represented by the set of complete plays following that strategy, *i.e.* as a *language* over the alphabet of moves. For a significant fragment of the programming language, the strategies arising as the interpretations of terms will turn out to be *regular languages*, which means that they can be represented by finite automata [20].

In fact, our preferred means of specification of strategies will be by using a certain class of *extended regular expressions*.

We recall firstly the standard syntax of regular expressions:

$$R \cdot S \quad R + S \quad R^* \quad a \quad \epsilon \quad 0$$

where a ranges over some alphabet Σ. These expressions are then interpreted as languages over Σ, i.e. as subsets of Σ^*, the set of all strings over Σ; we shall recall the basic definitions, while for more extensive background, we refer to standard texts such as [20].

Notation When we need to refer explicitly to the language denoted by a regular expression R, we shall write $\mathcal{L}(R)$.

We briefly recall the definitions: given $L, M \subseteq \Sigma^*$,

$$L \cdot M = \{st \mid s \in L \wedge t \in M\}$$

$$L^* = \bigcup_{i \in I\!N} L^i$$

where

$$L^0 = \{\epsilon\}, \quad L^{i+1} = L \cdot L^i.$$

Then

$$\begin{array}{lcl}
\mathcal{L}(R \cdot S) & = & \mathcal{L}(R) \cdot \mathcal{L}(S) \\
\mathcal{L}(R) + \mathcal{L}(S) & = & \mathcal{L}(R) \cup \mathcal{L}(S) \\
\mathcal{L}(R^*) & = & \mathcal{L}(R)^* \\
\mathcal{L}(a) & = & \{a\} \\
\mathcal{L}(\epsilon) & = & \{\epsilon\} \\
\mathcal{L}(0) & = & \varnothing
\end{array}$$

The extended regular expressions we will consider have the additional constructs

$$R \cap S \qquad \phi(R) \qquad \phi^{-1}(R)$$

where $\phi : \Sigma_1 \longrightarrow \Sigma_2^*$ is a homomorphism (more precisely, such a map uniquely induces a homomorphism between the free monoids Σ_1^* and Σ_2^*; note that, if Σ_1 is finite, such a map is itself a finite object). The interpretation of these constructs is the obvious one: $R \cap S$ is intersection of languages, $\phi(R)$ is direct image of a language under a homomorphism, and $\phi^{-1}(R)$ is inverse image.

$$\begin{array}{lcl}
\mathcal{L}(R \cap S) & = & \mathcal{L}(R) \cap \mathcal{L}(S) \\
\mathcal{L}(\phi(R)) & = & \{\phi(s) \mid s \in \mathcal{L}(R)\} \\
\mathcal{L}(\phi^{-1}(R)) & = & \{s \in \Sigma_1^* \mid \phi(s) \in \mathcal{L}(R)\}
\end{array}$$

The following is standard [20].

Proposition 2.1 *If we restrict to finite alphabets (so that in particular all homomorphisms map between finite alphabets) then every extended regular expression denotes a regular language, which can be recognized by a finite automaton which can be effectively constructed from the extended regular expression.*

We briefly indicate the arguments showing that regularity is preserved by the extended regular operations.

- $R \cap S$: a product automaton construction is used.
- $\phi(R)$: homomorphisms commute with regular expression operations.
- $\phi^{-1}(R)$: if (Q, q_0, F, δ) recognizes $\mathcal{L}(R)$, (Q, q_0, F, δ') recognizes $\mathcal{L}(\phi^{-1}(R))$, where:

$$\delta'(q, a) = q' \;\equiv\; \delta^*(q, \phi(a)) = q'.$$

Alphabet transformations Since we will interpret types as alphabets, getting the types right in our interpretation of terms as strategies will require transforming appropriately between different alphabets, and it is in this restricted form that homomorphisms will be used. The ideas are simple, but will be used repeatedly, and must be mastered at this stage.

We will "glue" types together using disjoint union:

$$X + Y = \{x^1 \mid x \in X\} \cup \{y^2 \mid y \in Y\}$$

We then have canonical maps arising from these disjoint unions.

$$\begin{tikzcd} X \arrow[r, shift left, "\mathsf{inl}"] & X + Y \arrow[l, shift left, "\mathsf{outl}"] \arrow[r, shift right, "\mathsf{outr}"'] & Y \arrow[l, shift right, "\mathsf{inr}"'] \end{tikzcd}$$

$$\begin{array}{ll} \mathsf{inl}(x) = x^1 & \mathsf{inr}(y) = y^2 \\ \mathsf{outl}(x^1) = x & \mathsf{outr}(x^1) = \epsilon \\ \mathsf{outl}(y^2) = \epsilon & \mathsf{outr}(y^2) = y \end{array}$$

Exercise Verify that

$$\mathsf{outl} \circ \mathsf{inl} = \mathsf{id}_X \qquad \mathsf{outr} \circ \mathsf{inr} = \mathsf{id}_Y.$$

What can you say about $\mathsf{outr} \circ \mathsf{inl}$? About $\mathsf{inl} \circ \mathsf{outl}$?

Example Shuffle Product: for regular expressions R and S over the alphabet Σ,

$$R \parallel S = \nabla(\mathsf{outl}^{-1}(R) \cap \mathsf{outr}^{-1}(S))$$

$$\begin{tikzcd} \Sigma \arrow[r, shift left, "\mathsf{inl}"] & \Sigma + \Sigma \arrow[l, shift left, "\mathsf{outl}"] \arrow[r, shift right, "\mathsf{outr}"'] & \Sigma \arrow[l, shift right, "\mathsf{inr}"'] \end{tikzcd}$$

$$\nabla : \Sigma + \Sigma \longrightarrow \Sigma$$

$$\nabla(a^1) = a = \nabla(a^2)$$

Exercise Verify that this expression does yield the shuffle product of the languages denoted by R and S, *i.e.* the set of "shuffles" or interleavings of strings drawn from $\mathcal{L}(R)$ and $\mathcal{L}(S)$.

The reader may wonder why we did not immediately stipulate that all alphabets are finite. This is because infinite alphabets do arise naturally in defining game semantics (for example, our game for the basic type $I\!N$ has infinitely many moves), so it is useful to consider this more general situation, even though we will lose effectiveness in general. (Note that the definitions of the meanings of extended regular expressions as formal languages still make sense even if the alphabet is infinite). For the same reason, we consider a further extension to the syntax of "regular expressions", even though it certainly does not preserve regularity in general. Namely, we consider infinite summation $\sum_{i \in I} R_i$, which of course is to be interpreted as union of a family of languages. (This extension is also used in Conway's classic treatise [9]).

What will be important is that for a significant fragment of the language whose semantics we will describe, the extended regular expressions (excluding infinite summations) over finite alphabets suffice, and hence Proposition 2.1 will apply.

2.7. DENOTATIONAL SEMANTICS À LA HOPCROFT AND ULLMAN

We now proceed to the formal description of the semantics.

Firstly, for each basic data type with set of data values D, we specify the following alphabet of moves:

$$M_D = \{q\} \cup D.$$

Note that $\mathbb{N}$ and **bool** follow this pattern. In each case, the idea is the same: O can initially use q to request a value, and the possible responses by P are drawn from the set D.

Note that $M_{\mathbb{N}}$ is an infinite alphabet. We will also consider finite truncations $M_{\mathbb{N}_k}$ where the set of data values is $\{0, \ldots, k-1\}$.

To interpret

$$x_1 : B_1, \ldots, x_k : B_k \vdash t : B$$

we will use the disjoint union of the alphabets:

$$M_{B_1} + \cdots + M_{B_k} + M_B$$

The "tagging" of the elements of the disjoint union to indicate which of the games $B_1, \ldots, B_k, B$ each move occurs in makes precise the "alignment by columns" of the moves in our informal displays of plays such as

$$\begin{array}{ccc} \mathbb{N} & \vdash & \mathbb{N} \\ & & q \\ q & & \\ n & & \\ & & n \end{array}$$

The corresponding play, as a word over the alphabet $M_{\mathbb{N}} + M_{\mathbb{N}}$, will now be written as

$$q^2 \cdot q^1 \cdot n^1 \cdot n^2.$$

Our general procedure is then as follows. Given a term in context

$$x_1 : B_1, \ldots, x_k : B_k \vdash t : B$$

we will give an extended regular expression

$$R = [\![x_1 : B_1, \ldots, x_k : B_k \vdash t : B]\!]$$

such that the language denoted by R is the strategy interpreting the term.

As a first example, for a constant $\vdash c : B$,

$$[\![\vdash c : B]\!] = q \cdot c.$$

The operation of addition, of type[1]

$$I\!N^1,\ I\!N^2 \vdash I\!N^3$$

is interpreted by

$$q^3 \cdot q^1 \cdot \sum_{n \in I\!N} (n^1 \cdot q^2 \cdot \sum_{m \in I\!N} (m^2 \cdot (n+m)^3))$$

Note that this extended regular expression is over an infinite alphabet, and involves an infinite summation.

Exercise Give a definition of addition for the truncated natural numbers $I\!N_k$. Note that this is not just a matter of restricting the infinite summations to finite ones. Verify that the resulting expression does denote a regular language.

Exercise Write down the regular expression giving the interpretation of the conditional $\mathbf{bool}, B, B \vdash B$ for $B = \mathbf{bool}$. Now use infinite summation to do the same for $B = I\!N$.

A variable

$$x : B^1 \vdash x : B^2$$

is interpreted by

$$q^2 \cdot q^1 \cdot \sum_{b \in V(B)} b^1 \cdot b^2$$

(here we use $V(B)$ for the set of data values in the basic type B). Next, we show how to interpret composition.

$$\frac{\Gamma \vdash t : A \quad x : A, \Delta \vdash u : B}{\Gamma, \Delta \vdash u[t/x] : B}$$

Our interpretation (of composition in this instance!) is, of course, *compositional*. That is, we assume we have already defined

$$R = [\![\Gamma \vdash t : A]\!] \qquad S = [\![x : A, \Delta \vdash u : B]\!]$$

as the intepretations of the premises in the rule for composition.

Next, we assemble the alphabets of the premises and the conclusion, and assign names to the canonical alphabet transformations relating them.

[1] Here and in subsequent examples, we tag the copies of the type to make it easier to track which component of the disjoint union—*i.e.* which "column"—each move occurs in.

$$
\begin{tikzcd}
M_\Gamma + M_A \arrow[r, shift left, "\mathsf{in}_1"] & M_\Gamma + M_A + M_\Delta + M_B \arrow[l, shift left, "\mathsf{out}_1"] \arrow[r, shift left, "\mathsf{out}_2"'] \arrow[d, shift right, "\mathsf{out}"'] & M_A + M_\Delta + M_B \arrow[l, shift left, "\mathsf{in}_2"'] \\
& M_\Gamma + M_\Delta + M_B \arrow[u, shift right, "\mathsf{in}"'] &
\end{tikzcd}
$$

The central type in this diagram combines the types of both the premises, including the type A, which will form the "locus of interaction" between t and u. The type of the conclusion arises from this type by "hiding" or erasing this locus of interaction, which thereby is made "internal" to the compound system formed by the composition. Thus the interpretation of composition is by "parallel composition plus hiding". Formally, we write

$$[\![\Gamma, \Delta \vdash u[t/x] : B]\!] = \mathsf{out}(\mathsf{out}_1^{-1}(R^*) \cap \mathsf{out}_2^{-1}(S)).$$

This algebraic expression may seem somewhat opaque: note that it is equivalent to the more intuitive expression

$$\{s/M_A \mid s/(M_\Delta + M_B) \in \mathcal{L}(R^*) \wedge s/M_\Gamma \in \mathcal{L}(S)\}$$

Here s/X means the string s with all symbols from X erased. This set expression will be recognised as essentially the definition of parallel composition plus hiding in the trace semantics for CSP [18]. Although less intuitive, the algebraic expression we gave as the "official" definition has the advantage of making it apparent that regular languages are closed under composition, and indeed of immediately yielding a construction of an automaton to recognise the resulting language.

2.8. COMMANDS

At this point, we have formal descriptions of all the programming constructs we have considered thus far. We shall now go on to complete our description of the semantics of a procedural language. What may come as a pleasant surprise is that the tools we have already developed can be easily extended to cover commands, local variables, and procedures.

Firstly, we consider commands. In our language, we have a basic type $\mathbf{com}$, with the following operations:

$$
\begin{array}{lll}
\mathbf{skip} & : & \mathbf{com} \\
\mathbf{seq} & : & \mathbf{com} \times \mathbf{com} \to \mathbf{com} \\
\mathbf{cond} & : & \mathbf{bexp} \times \mathbf{com} \times \mathbf{com} \to \mathbf{com} \\
\mathbf{while} & : & \mathbf{bexp} \times \mathbf{com} \to \mathbf{com}
\end{array}
$$

with the following, more colloquial equivalents:

$$\begin{array}{lcl}\mathbf{seq}(c_1, c_2) & \equiv & c_1; c_2 \\ \mathbf{cond}(b, c_1, c_2) & \equiv & \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \\ \mathbf{while}(b, c) & \equiv & \mathbf{while}\ b\ \mathbf{do}\ c\end{array}$$

For each operation

$$\omega : B_1 \times \cdots \times B_k \longrightarrow B$$

there is a typing rule

$$x_1 : B_1, \ldots, x_k : B_k \vdash \omega(x_1, \ldots, x_k) : B$$

which can be combined with the use of the composition rule to build up complex commands, just as we did for expressions.

The game interpreting the type **com** is extremely simple:

$$\begin{array}{c}\texttt{run} \\ | \\ \texttt{done}\end{array}$$

This can be thought of as a kind of "scheduler interface": the environment of a command has the opportunity to schedule it by playing the move `run`. When the command is finished, it returns control to the environment by playing `done`.

Formally, note that **com** is just another example of a basic type—in fact the *unit type*, with just one data value. Thus its alphabet of moves is

$$M_{\mathbf{com}} = \{\mathtt{run}, \mathtt{done}\}.$$

(We use `run` rather than q for descriptive effect).
The strategies interpreting the operations are also disarmingly simple. Firstly, note that **skip** is the unique constant for the unit type:

$$[\![\vdash \mathbf{skip} : \mathbf{com}]\!] = \mathtt{run} \cdot \mathtt{done}.$$

Now the following strategy interprets sequential composition.

$$\begin{array}{lccccc}\mathtt{seq}: & \mathbf{com} & \Rightarrow & \mathbf{com} & \Rightarrow & \mathbf{com} \\ & & & & & \mathtt{run} \\ & \mathtt{run} & & & & \\ & \mathtt{done} & & & & \\ & & & \mathtt{run} & & \\ & & & \mathtt{done} & & \\ & & & & & \mathtt{done}\end{array}$$

Formally, this is just

$$\mathbf{seq} : \mathbf{com}^1 \times \mathbf{com}^2 \to \mathbf{com}^3$$

$$[\![\mathbf{seq}]\!] = \mathtt{run}^3 \cdot \mathtt{run}^1 \cdot \mathtt{done}^1 \cdot \mathtt{run}^2 \cdot \mathtt{done}^2 \cdot \mathtt{done}^3.$$

Thus we think of sequential composition as working as follows. When asked to run by the environment, it begins by asking its first argument to run; when this argument responds, signalling that it has run to completion, it asks its second argument to run; when this has completed, it signals completion to its environment.

Exercise Give strategies to interpret $\mathbf{cond}$ and $\mathbf{while}$. These should be expressible as (non-extended) regular expressions.

Note that this way of modelling commands and their sequential composition is radically different to the traditional approach in denotational semantics (see e.g. [33, 32, 35, 34]), in which commands are modelled as *state transformers*, *i.e.* (roughly speaking) functions from states to states, and sequential composition as function composition. To fully understand the difference in our point of view, we must see how imperative variables are modelled in our approach; this is our next topic.

2.9. IMPERATIVE VARIABLES

The most distinctive part of an imperative language is of course the store upon which the commands operate. To interpret mutable variables, we will take an "object-oriented view" as advocated by John Reynolds [30]. In this view, a variable (say for example being used to store values of type $I\!N$) is seen as an object with two methods:

– the "read method", for dereferencing, giving rise to an operation of type $\mathtt{var} \Rightarrow I\!N$;
– the "write method", for assignment, giving an operation of type $\mathtt{var} \Rightarrow I\!N \Rightarrow \mathbf{com}$.

We *identify* the type of variables with the product of the types of these methods, setting

$$\mathtt{var} = (I\!N \Rightarrow \mathbf{com}) \times I\!N.$$

Now assignment and dereferencing are just the two projections, and we can interpret a command `x:=!x+1` as the strategy

$$
\begin{array}{ccccccc}
(\mathbb{N} & \Rightarrow & \mathbf{com}) & \times & \mathbb{N} & \Longrightarrow & \mathbf{com} \\
 & & & & & & \mathtt{run} \\
 & & & & \mathtt{read} & & \\
 & & & & n & & \\
 & & \mathtt{write} & & & & \\
q & & & & & & \\
n+1 & & & & & & \\
 & & \mathtt{ok} & & & & \\
 & & & & & & \mathtt{ok}
\end{array}
$$

(We use `write` and `ok` in place of `run` and `done` in the assignment part, and `read` in place of q in the dereferencing part, to emphasize that these moves initiate assignments and dereferencing rather than arbitrary commands or natural number expressions.)

In fact, we shall slightly simplify this description. We shall elide the opening two moves in a write operation, and simply have moves of the form $\mathbf{write}(d)$, for each possible value d of the data-type, and `ok` as the only possible reponse.

Thus our alphabet for the type $\mathbf{var}[D]$ of imperative variables which can have values from the set D stored in them, is given by

$$M_{\mathbf{var}[D]} = \{\mathtt{read}\} \cup D \cup \{\mathbf{write}(d) \mid d \in D\} \cup \{\mathtt{ok}\}$$

The operations for reading and writing have the form

$$
\begin{array}{llll}
\mathbf{assign} : \mathbf{var}[D] \times \mathbf{exp}[D] \to \mathbf{com} & \mathbf{assign}(v, e) & \equiv & v := e \\
\mathbf{deref} : \mathbf{var}[D] \to \mathbf{exp}[D] & \mathbf{deref}(v) & \equiv & !v
\end{array}
$$

The strategy for assign is:

$$\mathbf{assign} : \mathbf{var}[D]^1 \times \mathbf{exp}[D]^2 \to \mathbf{com}^3$$

$$[\![\mathbf{assign}]\!] = \mathtt{run}^3 \cdot q^2 \cdot \sum_{d \in D} (d^2 \cdot \mathbf{write}(d)^1) \cdot \mathtt{ok}^1 \cdot \mathtt{done}^3$$

Exercise Give the strategy for `deref`. Compute the strategy obtained for `x:=!x+1`.

2.9.1. *Block structure*

The key point is to interpret the *allocation* of variables correctly, so that if the variable x in the above example has been bound to a genuine storage cell, the various reads and writes made to it have the expected relationship. In general, a term M with a free variable x will be interpreted as a strategy for $\mathtt{var} \Rightarrow A$, where A is the type of M. We must interpret `new` x `in` M as a strategy for A by "binding x to a memory cell". With game semantics, this is easy! The strategy for M will play some moves in A, and may also make repeated use of the `var` part. The play in the `var` part will look something like this.

$$\begin{array}{c} \texttt{var} \\ \\ \texttt{write}(d_1) \\ \texttt{ok} \\ \texttt{write}(d_2) \\ \texttt{ok} \\ \texttt{read} \\ d_3 \\ \texttt{read} \\ d_4 \\ \vdots \end{array}$$

Of course there is nothing constraining the reads and writes to have the expected relationship. However, there is an obvious strategy

$$\mathbf{cell} : \texttt{var}$$

which plays like a storage cell, always responding to a `read` with the last value written. Once we have this strategy, we can interpret **new** by composition with **cell**, so

$$[\![\mathbf{new}\ x\ \mathbf{in}\ M]\!] = [\![M]\!] \circ \mathbf{cell}.$$

Two important properties of local variables are immediately captured by this interpretation:

Locality Since the action in `var` is hidden by the composition, the environment is unaware of the existence and use of the local variable.

Irreversibility As M interacts with `cell`, there is no way for M to undo any writes which it makes. Of course M can return the value stored in the cell to be the same as it has been previously, but only by performing a new `write`.

2.9.2. *Commands revisited*

We can now get a better perspective on how command combinators work. Suppose that we have a sequential composition

$$\frac{x : \texttt{var} \vdash c_1 : \mathbf{com} \qquad x : \texttt{var} \vdash c_2 : \mathbf{com}}{x : \texttt{var} \vdash c_1 ; c_2 : \mathbf{com}}$$

The game semantics of this command, formed by the composition of the **seq** combinator with the commands c_1 and c_2, can be pictured as follows.

$$
\begin{array}{cccccccc}
 & & \mathtt{var} \vdash & \mathbf{com} & \times & \mathbf{com} & \to & \mathbf{com} \\
 & & & & & & & \mathtt{run} \\
 & & & \mathtt{run} & & & & \\
c_1 & \vdots & & & & & & \\
 & & & \mathtt{done} & & & & \\
 & & & & & \mathtt{run} & & \\
c_2 & \vdots & & & & & & \\
 & & & & & \mathtt{done} & & \\
 & & & & & & & \mathtt{done}
\end{array}
$$

Note that the simple behaviour of the sequential composition combinator can be used, via composition of strategies, to sequence arbitrarily complex commands c_1 and c_2.

Exercise Compute the semantics of: `x:=0; x:=!x+1`.

This should be contrasted with the traditional denotational semantics of imperative state

$$\mathsf{State} = \mathsf{Val}^{\mathsf{Loc}}$$

in which states are modelled as mappings from locations to values, i.e. as "state snapshots". Programs are then modelled as state transformers, i.e. as functions or relations on these state snapshots, and sequential composition as function or relation composition. It turns out to be hard to give accurate models of locally scoped imperative state with these models; one has to introduce functor categories, and even then full abstraction is hard to achieve [27].

By contrast, we track the "stream of consciousness" of each variable as a separate, autonomous process. A similar approach to modelling variables has also been taken in other process models, for example by Milner in CCS [26]. However, the greater degree of structure provided by game semantics (the distinction between O and P, the constraints imposed by the rules of the game and so on) will enable us to obtain not merely a sound model but a fully abstract one.

3. The full language

We now give a more systematic account of the syntax and semantics of the full language we are considering, including (first-order) procedures.

3.1. SYNTAX

Firstly, we shall be more precise about the syntax of the language we are interpreting.

We assume given a family of basic data types D; these are just sets of data values, such as $I\!N$ and $\mathbf{bool}$. For each such data type, we will have types $\mathbf{exp}[D]$ and $\mathbf{var}[D]$ of expressions which can produce values of type D, and variables which can store values of type D respectively. We will also have a data type $\mathbf{com}$ of commands; this will really be $\mathbf{exp}[\mathbf{1}]$, where $\mathbf{1}$ is a one-element set, but it will be convenient to distinguish this special case. Thus our syntax of *basic types* is

$$B \quad ::= \quad \mathbf{exp}[D] \mid \mathbf{com} \mid \mathbf{var}[D].$$

The general class of types we shall consider will contain *first-order procedures* as well as basic types. The syntax is

$$T \quad ::= \quad B \mid B \to T.$$

The reason for the restriction to first-order is that it allows much simpler representations of the game semantics, and in particular makes effectivity much easier to achieve. The effective representation of the game semantics of programs at higher types is a topic of current research.

The language is an applied typed λ-calculus over this set of types. Typing judgements have the form

$$\Gamma \vdash t : T \qquad\qquad \Gamma = x_1 : T_1, \ldots, x_k : T_k$$

The type system is as follows.
Typed λ-calculus (I): Variables and Abstraction.

$$\frac{}{\Gamma, x : T \vdash x : T} \qquad\qquad \frac{\Gamma, x : B \vdash t : T}{\Gamma \vdash \lambda x.t : B \to T}$$

Typed λ-calculus (II): Application.

$$\frac{\Gamma \vdash t : B \to T \quad \Gamma \vdash u : B}{\Gamma \vdash tu : T}$$

Block structure:

$$\frac{\Gamma, x : \mathbf{var}[D] \vdash t : T}{\Gamma \vdash \mathbf{new}\ x\ \mathbf{in}\ t : B}$$

There is also a set of constants

$$\kappa : B_1 \to \cdots \to B_k \to B$$

as already described.

For the purposes of giving a semantics, it will be convenient to use a variant of the above system, in which the application rule is replaced by the following two rules:

Linear Application:

$$\frac{\Gamma \vdash t : B \to T \quad \Delta \vdash u : B}{\Gamma, \Delta \vdash tu : T}$$

Contraction:

$$\frac{\Gamma, x : U, y : U \vdash t : T}{\Gamma, z : U \vdash t[z/x, z/y] : T}$$

It is well know that the resulting system is equivalent, in the sense that exactly the same typing judgements can be derived.

Exercise Show how to derive (arbitrary instances of) the two new rules in the original system, and conversely how to derive any instance of the original rule for Application in the new system.

3.2. TYPES AS ALPHABETS

The first step in the game semantics is to interpret each type T by an alphabet of moves M_T. In fact, for basic types B it will be convenient to define sets Q_B and A_B (*questions* and *answers* of type B respectively), such that

$$Q_B \cap A_B = \varnothing, \qquad Q_B \cup A_B = M_B.$$

$$\begin{array}{lcllcl} Q_{\mathbf{exp}[D]} & = & \{q\} & \qquad A_{\mathbf{exp}[D]} & = & D \\ Q_{\mathbf{com}} & = & \{\mathtt{run}\} & \qquad A_{\mathbf{com}} & = & \{\mathtt{done}\} \\ Q_{\mathbf{var}[D]} & = & \{\mathtt{read}\} \cup \{\mathtt{write}(d) \mid d \in D\} & \qquad A_{\mathbf{var}[D]} & = & D \cup \{\mathtt{ok}\} \end{array}$$

We extend this to general types by

$$M_{B \to T} = M_B + M_T.$$

3.3. THE INTERPRETATION OF TERMS

The general format is as follows. A typing judgement

$$x_1 : T_1, \ldots, x_k : T_k \vdash t : T$$

is interpreted by a regular expression

$$R = [\![x_1 : T_1, \ldots, x_k : T_k \vdash t : T]\!]$$

over the alphabet

$$M_{T_1} + \cdots + M_{T_k} + M_T.$$

3.3.1. *Variables*
Let $T = B_1 \to \cdots B_k \to B$. The corresponding alphabet is $M_\Gamma + M^1_T + M^2_T$. Then $[\![\Gamma, x : T \vdash x : T]\!]$ is given by the following regular expression.

$$\sum_{q \in Q_B} q^2 \cdot q^1 \cdot (\sum_{i=1}^{k} R_i)^* \cdot \sum_{a \in A_B} (a^1 \cdot a^2)$$

where

$$R_i = \sum_{q' \in Q_{B_i}} q'^1 \cdot q'^2 \cdot \sum_{a' \in A_{B_i}} (a'^2 \cdot a'^1), \qquad 1 \leq i \leq k.$$

This is the copy-cat strategy for first-order procedure types. At such types the "generic behaviour" of a procedure is, on being called by its environment, to perform some sequence of calls of its arguments, and then to return a result. The behaviour of higher-order procedures can be much more complicated. See [28] for a description of the behaviours of second-order functions as deterministic context-free languages.

Exercise Try to give an interpretation of a second-order function variable, of type $(\mathbf{exp}[I\!N] \to \mathbf{exp}[I\!N]) \to \mathbf{exp}[I\!N]$. To get an idea of some of the complications which arise, it may be useful to consider the strategies corresponding to the following two terms:

$$\begin{aligned} M_1 &= \lambda f.\, f(\lambda x.\, f(\lambda y.\, y)) \\ M_2 &= \lambda f.\, f(\lambda x.\, f(\lambda y.\, x)) \end{aligned}$$

For a discussion of this example, see [7].

3.3.2. *Abstraction*

$$\frac{\Gamma, x : B \vdash t : T}{\Gamma \vdash \lambda x.\, t : B \to T}$$

If

$$[\![\Gamma, x : B \vdash t : T]\!] = R$$

then

$$[\![\Gamma \vdash \lambda x.\, t : B \to T]\!] = \phi(R)$$

where ϕ is the trivial associativity isomorphism for disjoint union:

$$\phi : (M_\Gamma + M_B) + M_T \xrightarrow{\cong} M_\Gamma + (M_B + M_T)$$

3.3.3. *Linear Application*

This follows very similar lines to our previous treatment of composition (as one would expect).

$$\frac{\Gamma \vdash t : B \to T \quad \Delta \vdash u : B}{\Gamma, \Delta \vdash tu : T}$$

$$R = [\![\Gamma \vdash t : B \to T]\!] \qquad S = [\![\Delta \vdash u : B]\!]$$

$$\begin{tikzcd}
M_\Gamma + M_B + M_T \arrow[r, shift left, "\mathsf{in}_1"] & M_\Gamma + M_\Delta + M_B + M_T \arrow[l, shift left, "\mathsf{out}_1"] \arrow[r, shift right, "\mathsf{out}_2"'] \arrow[d, shift right, "\mathsf{out}"'] & M_\Delta + M_B \arrow[l, shift right, "\mathsf{in}_2"'] \\
 & M_\Gamma + M_\Delta + M_T \arrow[u, shift right, "\mathsf{in}"'] &
\end{tikzcd}$$

$$[\![\Gamma, \Delta \vdash tu : T]\!] = \mathsf{out}(\mathsf{out}_1^{-1}(R) \cap \mathsf{out}_2^{-1}(S^*)).$$

3.3.4. *Contraction*

If $R = [\![\Gamma, x : T, y : T \vdash u : U]\!]$, then

$$[\![\Gamma, z : T \vdash u[z/x, z/y] : U]\!] = \mathsf{id}_{M_\Gamma} + \nabla + \mathsf{id}_{M_U}(R)$$

where $\nabla : M_T + M_T \longrightarrow M_T$. This simply "de-tags" moves from the two occurrences x and y so that they both come from the same occurrence z. It is a property of first-order types—but *not* of higher-order ones—that this de-tagging results in an unambiguous description, in which the "threads" of interaction involving the two occurrences can still be disentangled.

3.3.5. *Local variables*

$$\frac{\Gamma, x : \mathbf{var}[D] \vdash t : T}{\Gamma \vdash \mathbf{new}\ x\ \mathbf{in}\ t : T}$$

Let

$$\begin{array}{lcl} R & = & [\![\Gamma, x : \mathbf{var}[D] \vdash t : T]\!], \\ \mathbf{cell}_D & = & (\sum_{d \in D}(\mathtt{write}(d) \cdot \mathtt{ok} \cdot (\mathbf{read} \cdot d)^*))^* \end{array}$$

Then

$$[\![\Gamma \vdash \mathbf{new}\ x\ \mathbf{in}\ t : T]\!] = \mathsf{out}_2(R \cap \mathsf{out}_1^{-1}(\mathbf{cell}_D))$$

where

$$\begin{tikzcd}
M_{\mathbf{var}[D]} \arrow[r, shift left, "\mathsf{in}_1"] & M_\Gamma + M_{\mathbf{var}[D]} + M_T \arrow[l, shift left, "\mathsf{out}_1"] \arrow[r, shift right, "\mathsf{out}_2"'] & M_\Gamma + M_T \arrow[l, shift right, "\mathsf{in}_2"']
\end{tikzcd}$$

3.4. THE FINITARY SUB-LANGUAGE

We define the finitary fragment of our procedural language to be that in which only *finite* sets of basic data values D are used. For example, we may consider only the basic types over $\mathbf{bool}$ and $\mathbb{N}_k$, omitting $\mathbb{N}$.

The following result is immediate from the above definitions.

Proposition 3.1 *The types in the finitary language are intepreted by finite alphabets, and the terms are interpreted by extended regular expressions without infinite summations. Hence terms in this fragment denote regular languages.*

4. Model checking

Terms M and N are defined to be *observationally equivalent*, written $M \equiv N$, just in case for any context $C[\cdot]$ such that both $C[M]$ and $C[N]$ are programs (i.e. closed terms of type $\mathbf{com}$), $C[M]$ converges (i.e. evaluates to $\mathbf{skip}$) if and only if $C[N]$ converges. (Note that the quantification over all *program* contexts takes side effects of M and N fully into account; see e.g. [5].) The theory of observational equivalence is rich; for example, here are some non-trivial observationally equivalences (Ω is the divergent program):

$$P : \mathbf{com} \vdash \ \mathbf{new}\ x\ \mathbf{in}\ P \ \equiv\ P \tag{1}$$

$$P : \mathbf{com} \to \mathbf{com} \vdash \ \begin{array}{l} \mathbf{new}\ x{:=}0\ \mathbf{in}\ P(x{:=}1); \\ \mathbf{if}\ !x = 1\ \mathbf{then}\ \Omega\ \mathbf{else\ skip} \\ \equiv \\ P(\Omega) \end{array} \tag{2}$$

$$P : \mathbf{com} \to \mathbf{com} \vdash \ \begin{array}{l} \mathbf{new}\ x{:=}0\ \mathbf{in} \\ P(x{:=}!x + 2);\ \mathbf{if}\ \mathbf{even}(x)\ \mathbf{then}\ \Omega \\ \equiv \\ \Omega \end{array} \tag{3}$$

$$P : \mathbf{com} \to \mathbf{bexp} \to \mathbf{com} \vdash \ \begin{array}{l} \mathbf{new}[\mathbf{int}]\ x{:=}1\ \mathbf{in}\ P(x{:=}-x)(x > 0) \\ \equiv \\ \mathbf{new}[\mathbf{bool}]\ x{:=}\mathbf{true}\ \mathbf{in}\ P(x{:=}\neg x)\, x \end{array} \tag{4}$$

Exercise Try to reason informally about the validity of these equivalences.

Theorem 1 ([5, 7, 15])

$$\Gamma \vdash t \equiv u \iff \mathcal{L}(R) = \mathcal{L}(S)$$

where $R = [\![\Gamma \vdash t : T]\!]$, $S = [\![\Gamma \vdash u : T]\!]$.

Moreover, $\mathcal{L}(R) = \mathcal{L}(S)$ (equality of (languages denoted by) regular expressions) is decidable. Hence observation equivalence for the finitary sub-language is decidable.

If $R \neq S$ we will obtain a witness $s \in [\![R]\!] \triangle [\![S]\!]$, which we can use to construct a separating context $C[\cdot]$, such that

$$\mathsf{eval}(C[t]) \neq \mathsf{eval}(C[u]).$$

4.1. MODEL CHECKING BEHAVIOURAL PROPERTIES

The same algorithmic representations of program meanings which are used in deciding observational equivalence can be put to use in verifying a wide range of program properties, and in practice this is where the interesting applications are most likely to lie. The basic idea is very simple. To verify that a term-in-context $\Gamma \vdash M : A$ satisfies behavioural property $\phi \subseteq M^*_{\Gamma,A}$ amounts to checking $[\![\Gamma \vdash M : A]\!] \subseteq \phi$, which is decidable if ϕ is, for example, regular. Such properties can be specified in temporal logic, or simply as regular expressions.

As a first example of such a property, consider the sequent

$$x : \mathbf{var}[D], p : \mathbf{com} \vdash \mathbf{com}.$$

Suppose we wish to express the property

"x is written before p is (first) called".

The alphabet of the sequent is

$$M = M^1_{\mathbf{var}[D]} + M^2_{\mathbf{com}} + M^3_{\mathbf{com}}.$$

The following regular expression captures the required property:

$$X^* \cdot \sum_{d \in D} \mathtt{write}(d)^1 \cdot X^* \cdot \mathtt{run}^2 \cdot M^*$$

where $X = M \setminus M^2_{\mathbf{com}}$.

As a more elaborate example, consider the sequent

$$p : \mathbf{exp}[D] \rightarrow \mathbf{exp}[D], x : \mathbf{var}[D] \vdash \mathbf{com}$$

and the property:

"whenever p is called, its argument is read from x, and its result is immediately written into x".

This time, the alphabet is

$$M = (M^1_{\mathbf{exp}[D]} + M^2_{\mathbf{exp}[D]}) + M^3_{\mathbf{var}[D]} + M^4_{\mathbf{com}}$$

and the property can be captured by the regular expression

$$(X^* \cdot (q^1 \cdot \mathtt{read}^3 \cdot \sum_{d \in D} (d^3 \cdot d^1) \cdot Y^* \cdot \sum_{d \in D} (d^2 \cdot \mathtt{write}(d)^3) \cdot \mathtt{ok}^3 \cdot Z^*)^*)^*$$

for suitable choices of sets of moves X, Y, Z.

Exercise Find suitable choices for X, Y and Z.

Exercise Find more interesting properties!

This example illustrates the inherent compositionality of our approach, being based on a compositional semantics which assigns meanings to terms-in-context.

Our approach combines gracefully with the standard methods of *over-approximation* and *data-abstraction*. The idea of over-approximation is simple and general:

$$[\![\Gamma, \vdash M : A]\!] \subseteq S \ \wedge \ S \subseteq \phi \implies [\![\Gamma, \vdash M : A]\!] \subseteq \phi.$$

This means that we can "lose information" in over-approximating the semantics of a program while still inferring useful information about it. This combines usefully with the fact that all the regular expression constructions used in our semantics are *monotone*, which means that if we over-approximate the semantics of some subterms $t_1, \ldots, t_n$, and calculate the meaning of the context $C[\cdot, \ldots, \cdot]$ in which the sub-terms are embedded in the standard way, then the resulting interpretation of $t = C[t_1, \ldots, t_n]$ will over-approximate the "true" semantics $[\![t]\!]$.

An important and natural way in which over-approximation arises is from *data abstraction*. Suppose, for a simple example, that we divide the integer data type $\mathbb{Z}$ into "negative" and "non-negative". Since various operations (e.g. addition) will not be compatible with this equivalence relation, we must also add a set "negative *or* non-negative"—i.e. the whole of $\mathbb{Z}$. Now arithmetic operations can be defined to work on these three "abstract values". To define boolean-valued operations on these values, we must extend the type $\mathbf{bool}$ with "true *or* false", which we write as ?. These extended booleans must in turn be propagated through conditionals and loops, which we do using the *non-determinism* which is naturally present in our regular expression formalism. For example, the conditional of type

$$\mathbf{exp}[\mathbf{bool}]^1 \to \mathbf{exp}[\mathbf{bool}]^2 \to \mathbf{exp}[\mathbf{bool}]^3 \to \mathbf{exp}[\mathbf{bool}]^4$$

can be defined thus:

$$q^4 \cdot q^1 \cdot (\mathbf{true}^1 \cdot R \ + \ \mathbf{false}^1 \cdot S + ?^1 \cdot (R + S))$$

where

$$R = q^2 \cdot \sum_{b \in \mathbf{bool}} (b^2 \cdot b^4), \qquad\qquad S = q^3 \cdot \sum_{b \in \mathbf{bool}} (b^3 \cdot b^4).$$

This over-approximates the meaning in the obvious way (which is of course quite classical in flow analysis): if we don't know whether the boolean value used to control the conditional is really true or false, then *we take both branches*. We can then use the monotonicity properties of the semantics to compute the interpretations of the λ-calculus constructs as usual, and conclude that the meaning assigned to the whole term over-approximates the "true" meaning, and hence that properties inferred of the abstraction hold for the original program. This gives an attractive approach to many of the standard issues in program analysis, e.g. inter-procedural control-flow analysis and reachability analysis [29, 8].

Of course, all of this fits into the framework of *abstract interpretation* [10] in a very natural way.

Another extant technique which can be nicely adapted to our setting is *data independence*. A program is said to be data independent with respect to a data type T if the only operations involving T it can perform are to input, output, and assign values of type T, as well as to test pairs of such values for equality. The impressive results on data independence in [23, 24] include sufficient conditions for reducing the verification of properties that are universally quantified over all instantiations of the data types, to the verification of the same properties for a finite number of finite instantiations.

In conclusion, this area seems ripe for further development, and to have the potential to act as a very effective bridge between semantics and computer-assisted verification and program analysis.

References

1. S. Abramsky. Semantics of interaction: an introduction to game semantics. In *Semantics and Logics of Computation*, pages 1–32. Cambridge Univ. Press, 1997.
2. S. Abramsky, K. Honda, and G. McCusker. Fully abstract game semantics for general references. In *Proceedings of IEEE Symposium on Logic in Computer Science, 1998*, pages 334–344. Computer Society Press, 1998.
3. S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. *J. Symb. Logic*, 59:543–574, 1994.
4. S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Information and Computation*, 163:409–470, 2000.
5. S. Abramsky and G. McCusker. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. In P. W. O'Hearn and R. D. Tennent, editors, *Algol-like languages*, pages 297–330. Birkhaüser, 1997.
6. S. Abramsky and G. McCusker. Call-by-value games. In *Proceedings of CSL '97*, number 1414 in Lecture Notes in Computer Science, pages 1–17. Springer-Verlag, 1998.

7. S. Abramsky and G. McCusker. Game semantics. In H. Schwichtenberg and U. Berger, editors, *Computational Logic: Proceedings of the 1997 Marktoberdorf Summer School*, pages 1–56. Springer-Verlag, 1998.
8. T. Ball and S. K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, MicroSoft Research, 2000.
9. J. H. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, 1971.
10. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of 4th ACM Symp. POPL*, pages 238–252, 1977.
11. V. Danos and R. Harmer. Probabilistic game semantics. In *Proc. IEEE Symposium on Logic in Computer Science, Santa Barbara, June, 2000*. Computer Science Society, 2000.
12. J. Corbett *et al*. Bandera: Extracting finite-state models from java source code. In *Proceedings of the 2000 International Conference on Software Engineering*, 2000.
13. D. R. Ghica. A regular-language model for Hoare-style correctness statements. In *Proc. 2nd Int. Workshop on Verification and Computational Logic VCL'2001, Florence, Italy*, 2001. `www.cs.queensu.ca/home/ghica/`.
14. D. R. Ghica. Regular language semantics for a call-by-value programming language. In *Proc. 17th Conf. Mathematical Foundations of Programming Semantics, Aarhus, Denmark*, 2001. `www.cs.queensu.ca/home/ghica/`.
15. 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*, pages 103–116. Springer-Verlag, 2000. LNCS Vol. 1853.
16. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989. Cambridge Tracts in Theoretical Computer Science 7.
17. R. Harmer and G. McCusker. A fully abstract game semantics for finite nondeterminism. In *Proceedings of Fourteenth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1999.
18. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
19. K. Honda and N. Yoshida. Game-theoretic analysis of call-by-value computation (extended abstract). In *Proc. of ICALP'97, Borogna, Italy, July, 1997*, LNCS. Springer-Verlag, 1997.
20. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
21. J. M. E. Hyland and C.-H. L. Ong. On Full Abstraction for PCF: I. Models, observables and the full abstraction problem, II. Dialogue games and innocent strategies, III. A fully abstract and universal game model. *Information and Computation*, 163:285–408, 2000.
22. J. Laird. *A semantic analysis of control*. PhD thesis, University of Edinburgh, 1998.
23. R. Lazic and D. Nowak. A unifying approach to data-independence. In *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR 2000)*. Springer-Verlag, 2000. LNCS.
24. R. S. Lazic, T. C. Newcomb, and A. W. Roscoe. On model checking data-independent systems with arrays without reset (abstract). In *Proceedings of VCL 2001*, 2001.
25. P. Malacaria and C. Hankin. Non-deterministic games and program analysis: an application to security. In *Proceedings of 14th Annual IEEE Symp. Logic in Computer Science*, pages 443–452. IEEE Computer Society, 1999.
26. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
27. P. W. O'Hearn and R. D. Tennent. *Algol-like Languages: Volumes I and II*. Birkhäuser, 1997. Progress in Theoretical Computer Science.
28. C.-H. L. Ong. Equivalence of third order Idealized Algol is decidable. Technical report, Oxford University Computing Laboratory, 2001. In preparation.

29. T. Reps, S. Horowitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. ACM Symp. POPL*, pages 49–61, 1995.
30. J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North Holland, 1978.
31. D. A. Schmidt. On the need for a popular formal semantics. *ACM SIGPLAN Notices*, 32:115–116, 1997.
32. J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1979. The MIT Press Series in Computer Science.
33. C. Strachey. Fundamental concepts in programming languages. Lecture notes for the International Summer School in Computer Programming, Copenhagen, 1967.
34. R. D. Tennent. Denotational semantics. In S. Abramsky *et al*, editor, *Handbook of Logic in Computer Science*, pages 169–322. Oxford University Press, 1994.
35. G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993. Foundations of Computing Series.