Algorithmic Game Semantics and Software Model-Checking

Samson Abramsky
Oxford University Computing Laboratory

Background

- Game semantics has been successfully applied to modelling laguages with a range of computational features: (higher-order) procedures, locally-scoped imperative variables, control constructs (continuations, exceptions), non-determinism, probability, . . .
- Current work is turning in an algorithmic direction, with applications to software model-checking and program analysis in mind.

  Advantages of the approach:
  - Inherently compositional.
  - Soundness coming from the underlying semantic analysis.
  - Generality, good underlying mathematical structure.

Game Semantics: an informal introduction

Before proceeding to a detailed technical account, 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.

Basic points:

- Two-player games (P and O) - corresponding to *System* and *Environment* (or: *Term* and *Context*).
- O always moves first—the environment sets the system going—and thereafter the two players alternate.
- 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,

Modelling Values

q

$0 \quad 1 \quad 2 \quad \ldots$

and the strategy for 3 is "When O plays q, I will play 3."

$\mathbb{N}$	
q	O
3	P

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.

Expressions

Consider the expression

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

The game involved in evaluating this expression is formed from "three copies of $\mathbb{N}$", 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.

$\mathbb{N}$	,	$\mathbb{N}$	$\vdash$	$\mathbb{N}$	
				q	O
q					P
3					O
		q			P
		2			O
				5	P

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 $m + n$."

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

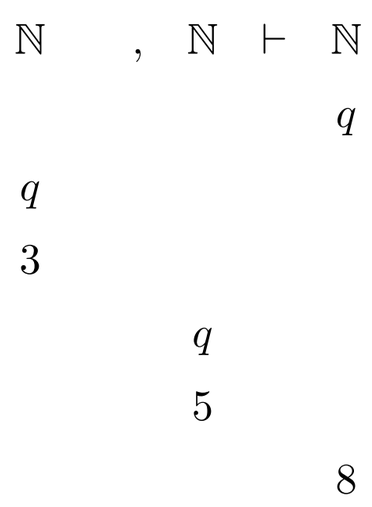
$$x : \mathbb{N}, y : \mathbb{N} \vdash x + y : \mathbb{N}$$

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.

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`.



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”.

Variables and copy-cat strategies

To interpret a variable

$$x : \mathbb{N} \vdash x : \mathbb{N}$$

we play as follows:

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

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 $\mathbb{N}$ to the other.

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.

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

Standard syntax of regular expressions

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

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

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

$$L^* = \bigcup_{i \in \mathbb{N}} L^i \quad \text{where} \quad L^0 = \{\epsilon\}, \quad L^{i+1} = L \cdot L^i.$$

Extended regular expressions

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.

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

Alphabet transformations

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.

$$X \underset{\mathsf{outl}}{\overset{\mathsf{inl}}{\rightleftarrows}} X + Y \underset{\mathsf{outr}}{\overset{\mathsf{inr}}{\rightleftarrows}} Y$$

$$\mathsf{inl}(x) = x^1 \qquad \mathsf{inr}(y) = y^2$$
$$\mathsf{outl}(x^1) = x \quad \mathsf{outr}(x^1) = \epsilon$$
$$\mathsf{outl}(y^2) = \epsilon \quad \mathsf{outr}(y^2) = y$$

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 $\mathbf{bool}$ follow this pattern.

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

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

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 $\vdash c : B$,

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

Addition revisited

The operation of addition, of type[a]

$$\mathbb{N}^1,\ \mathbb{N}^2 \vdash \mathbb{N}^3$$

is interpreted by

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

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

[a] 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.

Variables

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).

Composition

Next, we show how to interpret composition.

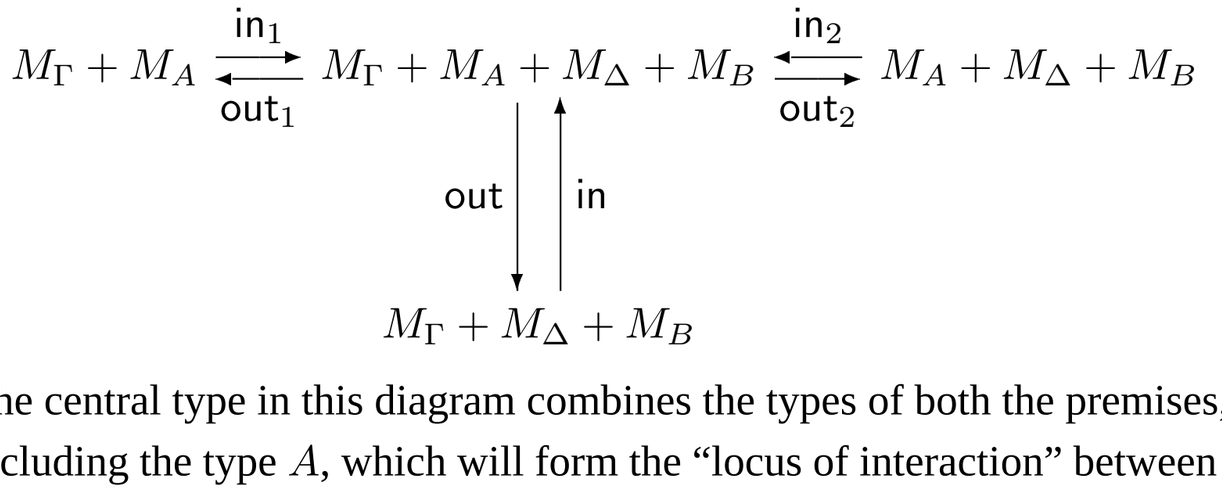
$$\frac{\Gamma \vdash t : A \quad x : A, \Delta \vdash u : B}{\Gamma, \Delta \vdash u[t/x] : B}$$

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.



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. 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.

Commands

Firstly, we consider commands. In our language, we have a basic type **com**, with the following operations:

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

More colloquial equivalents:

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

The game interpreting the type $\mathbf{com}$ is extremely simple:

$$\texttt{run}$$

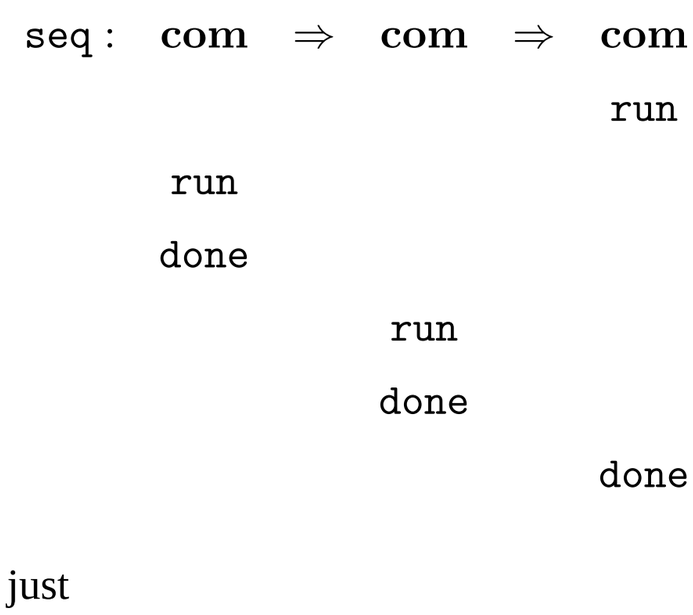
$$\texttt{done}$$

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 $\texttt{run}$. When the command is finished, it returns control to the environment by playing $\texttt{done}$.

The strategies interpreting the operations are also disarmingly simple. Firstly, note that $\mathbf{skip}$ is the unique constant for the unit type:

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

Now the following strategy interprets sequential composition.



Formally, this is just

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

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

Imperative variables

To interpret mutable variables, we will take an "object-oriented view" as advocated by John Reynolds. In this view, a variable (say for example being used to store values of type $\mathbb{N}$) is seen as an object with two methods:

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

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

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

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

In fact, we shall slightly simplify this description. 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

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

The strategy for assign is:

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

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

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 $\mathtt{new}\ x\ \mathtt{in}\ M$ 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 $\mathtt{var}$ part.

The play in the `var` part will look something like this.

`var`

`write`(d_1)

`ok`

`write`(d_2)

`ok`

`read`

d_3

`read`

d_4

$\vdots$

Of course there is nothing constraining the reads and writes to have the expected relationship. However, there is an obvious strategy $\mathbf{cell} : \mathtt{var}$:

$$\mathbf{cell} = (\sum_{n \in \mathbb{N}} (\mathtt{write}(n) \cdot \mathtt{ok} \cdot (\mathtt{read} \cdot n)^*))^*$$

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`.

Commands revisited

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

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

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


var ⊢ com × com → com
run
run
c_1
done
run
c_2
done
done

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.

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. See Algol-like Languages, ed. O'Hearn and Tennent, Birkhauser 1997.

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 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.

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.) The theory of observational equivalence is rich; for example, here are some non-trivial observationally equivalences (Ω is the divergent program):

Observation equivalences for Algol-like programs

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

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

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

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

Theorem 1

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

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

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

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.

Example

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:

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

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

As a more elaborate example, consider the sequent

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

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.

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 sub-terms $t_1, \ldots, t_n$, 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

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

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.

Of course, all of this fits into the framework of *abstract interpretation* (P. and R. Cousot) in a very natural way.

There is an on-going research project at Oxford University Computing Laboratory on Algorithmic Game Semantics with Luke Ong, Dan Ghica and Andrzej Murawski.

Some references:

- S.A. ‘Algorithmic Game semantics: a tutorial introduction’. In Proceedings of Marktoberdorf 2001. (Also available from my home page).
- Dan Ghica and Guy McCusker. ‘Reasoning about Idealized Algol using Regular Languages’. In Proceedings of ICALP 2000.
- Luke Ong. ‘Observation equivalence for third-order Idealized Algol is undecidable’. In Proceedings of LiCS 2002.