An introduction to algorithmic game semantics

Guy McCusker

University of Bath

GAMES 2009, Udine

Some questions

Consider a simple programming language including

- basic while-programs: variable assignment and lookup, arithmetic, while-loops
- first-order procedures
- local variables

over a *finite data set*.

Say that two program phrases M and N are *equivalent* if M can always be replaced by N wherever it appears in a program, without changing the behaviour.

Questions

- Is program equivalence decidable?
- If so, what is its complexity?

A semantic approach

One way to attack a question like this is via denotational semantics:

- construct a mathematical model of the language which captures the appropriate notion of equivalence
- study the decidability and complexity properties of the model.

In this talk, we will see how *game semantics* allows us to do just this.

Game Semantics

- A form of denotational semantics.
- Models computation or proof as *interaction* between a system and its environment.
- A program/proof is modelled as a *set of possible interactions*.

Games form a very rich, expressive universe which can model a variety of logics and programming languages very accurately.

Algorithmic Game Semantics

- The name given by Abramsky to the study of decidability and complexity properties of games models.
- Initiated by Ghica and McCusker in ICALP 2001's *Reasoning about Idealised Algol using regular languages.*
- Developed much further by Abramsky, Ghica, Murawski, Ong and Walukiewicz.

Prehistory of Game Semantics

The earliest precursor of game semantics is in the games-based interpretations of logic studied by Lorenzen, Lorenz et al.

Proofs are modelled as dialogues between two characters: the *verifier* V and the *falsifier* F.

For example, consider

$$\forall x.\exists y.x < y \wedge \mathsf{prime}(y).$$

A dialogue

F tries to show the statement false by picking a value for x which renders the statement

$$\exists y. x < y \wedge \mathsf{prime}(y)$$

false.

V responds by picking a value for y which he claims makes

$$x < y \wedge \mathsf{prime}(y)$$

true.

F tries to show this false by picking one side of the conjunction which he claims does not hold, and so on...

HO game semantics

We will be concerned with games in the style of Hyland and Ong. Such games models have been used to build models for higher-order programming languages with a variety of computational features:

- pure functional languages (PCF)
- mutable store (Idealized Algol)
- control operators (SPCF, exceptions)
- higher-order store (pointers)
- nondeterminism
- concurrency

Games

Computation = interaction between Opponent (O) and Player(P).

O	vs	P
Environment		System
Context		Program

Example: a first-order function

In a first-order function, the system *consumes* input and *produces* output, while the environment *produces* input and *consumes* output.

A typical interaction of the successor function with its environment might look like this.

- O: what is the output of this function?

Example: a first-order function

In a first-order function, the system *consumes* input and *produces* output, while the environment *produces* input and *consumes* output.

A typical interaction of the successor function with its environment might look like this.

- O: what is the output of this function?
- P: what is the input to this function?

Example: a first-order function

In a first-order function, the system *consumes* input and *produces* output, while the environment *produces* input and *consumes* output.

A typical interaction of the successor function with its environment might look like this.

- O: what is the output of this function?
- P: what is the input to this function?
- O: the input is 3.

Example: a first-order function

In a first-order function, the system *consumes* input and *produces* output, while the environment *produces* input and *consumes* output.

A typical interaction of the successor function with its environment might look like this.

- O: what is the output of this function?
- P: what is the input to this function?
- O: the input is 3.
- P: the output is 4.

Higher-Order Functions

The same ideas extend to model higher-order programs.

$$f : \mathbb{N} \rightarrow \mathbb{N} \vdash f(3) : \mathbb{N}$$

- O: what is the result of this program?

Higher-Order Functions

The same ideas extend to model higher-order programs.

$$f : \mathbb{N} \to \mathbb{N} \vdash f(3) : \mathbb{N}$$

- O: what is the result of this program?
- P: what is the output of the function f?

Higher-Order Functions

The same ideas extend to model higher-order programs.

$$\mathrm{f} : \mathbb{N} \rightarrow \mathbb{N} \vdash \mathrm{f}(3) : \mathbb{N}$$

- O: what is the result of this program?
- P: what is the output of the function f?
- O: what is the input to f?

Higher-Order Functions

The same ideas extend to model higher-order programs.

$$f : \mathbb{N} \to \mathbb{N} \vdash f(3) : \mathbb{N}$$

- O: what is the result of this program?
- P: what is the output of the function f?
- O: what is the input to f?
- P: the input to f is 3.

Higher-Order Functions

The same ideas extend to model higher-order programs.

$$f : \mathbb{N} \rightarrow \mathbb{N} \vdash f(3) : \mathbb{N}$$

- O: what is the result of this program?
- P: what is the output of the function f?
- O: what is the input to f?
- P: the input to f is 3.
- O: the output of f is 4.

Higher-Order Functions

The same ideas extend to model higher-order programs.

$$\mathtt{f} : \mathbb{N} \to \mathbb{N} \vdash \mathtt{f}(3) : \mathbb{N}$$

- O: what is the result of this program?
- P: what is the output of the function $\mathtt{f}$?
- O: what is the input to $\mathtt{f}$?
- P: the input to $\mathtt{f}$ is 3.
- O: the output of $\mathtt{f}$ is 4.
- P: the result of the program is 4.

Abbreviated notation

We will draw such dialogues as follows.

$$
\begin{array}{ccccccc}
f: & \mathbb{N} & \rightarrow & \mathbb{N} & \vdash f(3): & \mathbb{N} \\
 & & & & & q \\
 & & & q & & \\
 & q & & & & \\
 & 3 & & & & \\
 & & & 4 & & \\
 & & & & & 4
\end{array}
$$

Abbreviated notation

The move q is a request for data, and the number-moves are the supply of data.

Moves are written below the part of the type to which they correspond.

Time flows downwards.

Note how the O/P roles of the moves relating to f are the reverse of the situation for the successor function.

Multiple use of arguments

Programs may use their arguments more than once:

$f:$	$\mathbb{N}$	$\rightarrow$	$\mathbb{N}$	$\vdash f(3) + f(0):$	$\mathbb{N}$
					q
			q		
	q				
	3				
			n		
			q		
	q				
	0				
			m		
					$m + n$

Nested use of arguments

Programs may nest uses of their arguments:

$f:$	$\mathbb{N}$	$\to$	$\mathbb{N}$	$\vdash f(f(3)):$	$\mathbb{N}$
					q
			q		
	q				
			q		
	q				
	3				
			n		
	n				
			m		
					m

Not so simple...

We actually need more data than just the moves that are played.
Consider the terms

$$f : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \vdash \quad f(\lambda x.f(\lambda y.y))$$
$$\text{and} \quad f(\lambda x.f(\lambda y.x))$$

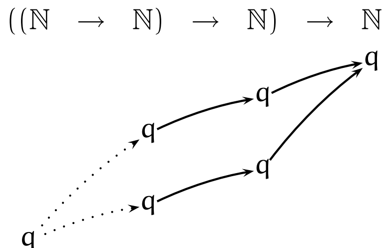
Different terms, same plays

The games interpretations of these terms will both play as follows:

$$
\begin{array}{ccccccc}
((\mathbb{N} & \rightarrow & \mathbb{N}) & \rightarrow & \mathbb{N}) & \rightarrow & \mathbb{N} \\
 & & & & & & q \\
 & & & & q & & \\
 & & q & & & & \\
 & & & & q & & \\
 & & q & & & & \\
q & & & & & &
\end{array}
$$

Different terms, same plays

The Hyland-Ong approach to resolving this ambiguity is to augment moves with *justification pointers*:



We will restrict our attention to fragments of the model where these pointers do not matter much, and so we ignore them in this lecture.

A definition (almost)

A *game* is

- a set of *moves* M
- a *labelling function* $\lambda : M \to \{O, P\}$
- further data which define (among other things) a set $L \subseteq M^*$ of *legal plays*.

In a legal play:

- O goes first
- P and O take turns to move thereafter (the play is an *alternating sequence* of moves).

Other rules can be imposed, depending on what kind of programs you want to model.

Types denote games

In a programming language, *types* are constraints on programs.

To say that a program P has type A implies that P's behaviour is of a certain kind.

Games prescribe a range of possible interactions between player and opponent.

Types will be interpreted as games.

Programs denote strategies

A *program* is modelled as a *set of behaviours* in the game corresponding to the program's type. This is what a *strategy* is.

A strategy σ for a game A is a set of legal plays of A such that:

- if $s \in \sigma$ then s has even length: we only record those behaviours where P has just responded to O's move
- $saa' \in \sigma \Rightarrow s \in \sigma$
- $saa' \in \sigma \wedge saa'' \in \sigma \Rightarrow saa' = saa''$.

More constraints can be applied; we will gloss over these.

Who wins? Who cares?

We are *not* concerned with who wins a game.

Winning corresponds to some kind of *totality* condition, saying that e.g. the system can always respond. We wish to model *all* programs, including nonterminating ones, so we ignore winning.

(On the other hand, when interpreting logic, winning is a central concern: one only wants complete proofs.)

Building games: basic types

The game interpreting a base type like the natural numbers is a two-move affair.

Opponent begins by asking for a number, and player may respond with any number.

$$\begin{aligned} M_{\mathbb{N}} &= \{q, 0, 1, 2, \ldots\} \\ \lambda_{\mathbb{N}}(q) &= O \\ \lambda_{\mathbb{N}}(n) &= P \end{aligned}$$

The only legal plays are those of the form

$$q \cdot n.$$

Building games: function types

Given games A and B we define $A \Rightarrow B$ as follows.

$$\begin{array}{rcl} M_{A \Rightarrow B} & = & M_A + M_B, \qquad \text{disjoint union} \\ \lambda_{A \Rightarrow B}(b) & = & \lambda_B(b) \\ \lambda_{A \Rightarrow B}(a) & = & \begin{cases} O, \text{ if } \lambda_A(a) = P \\ P, \text{ if } \lambda_A(a) = O \end{cases} \end{array}$$

A legal play of $A \Rightarrow B$ is an alternating sequence s such that

- $s \upharpoonright B$ is legal in B
- $s \upharpoonright A$ is an interleaving of legal A-plays, and is itself an alternating sequence.

Building games: product types

Given games A and B we define $A \times B$ as follows.

$$
\begin{array}{rcl}
M_{A\times B} & = & M_A + M_B \\
\lambda_{A\times B}(a) & = & \lambda_A(a) \\
\lambda_{A\times B}(b) & = & \lambda_B(b)
\end{array}
$$

A legal play of $A \times B$ is an alternating sequence s such that

- $s \upharpoonright A$ is legal in A
- $s \upharpoonright B$ is legal in B

Interpreting programs

A program of the form

$$x : A, y : B \vdash P : C$$

will be interpreted as a strategy for the game

$$A \times B \Rightarrow C$$

An example

Here is a play which might arise in the strategy for the addition program

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

$$\begin{array}{ccccc} \mathbb{N} & \times & \mathbb{N} & \Rightarrow & \mathbb{N} \\ & & & & q \\ q & & & & \\ 3 & & & & \\ & & q & & \\ & & 4 & & \\ & & & & 7 \end{array}$$

Currying

Compare the previous play to the curried version:

$\mathbb{N}$	$\Rightarrow$	$(\mathbb{N}$	$\Rightarrow$	$\mathbb{N})$
				q
q				
3				
		q		
		4		
				7

They're identical! This is important since it will let us interpret λ-abstraction...

A category of games

We can now build a category:

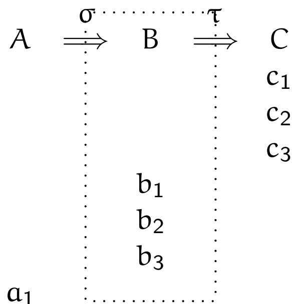
Objects:	games
Maps $A \to B$:	strategies for $A \Rightarrow B$

This will turn out to be a cartesian closed category, hence a model of λ-calculus.

We can model specific programming languages by giving an interpretation of their base types as games, and their constants as strategies.

Composition of strategies

"Parallel composition plus hiding".



Multiple uses of arguments

What happens when we try to compose the successor function

$$
\begin{array}{ccccc}
I & \longrightarrow & (\mathbb{N} & \Rightarrow & \mathbb{N}) \\
 & & & & q \\
 & & q & & \\
 & & n & & \\
 & & & & (n+1)
\end{array}
$$

with the strategy for $f(f(3))$?

A reminder of $f(f(3))$

$f:$	$\mathbb{N}$	$\rightarrow$	$\mathbb{N}$	$\vdash f(f(3)):$	$\mathbb{N}$
					q
			q		
	q				
			q		
	q				
	3				
			n		
	n				
			m		
					m

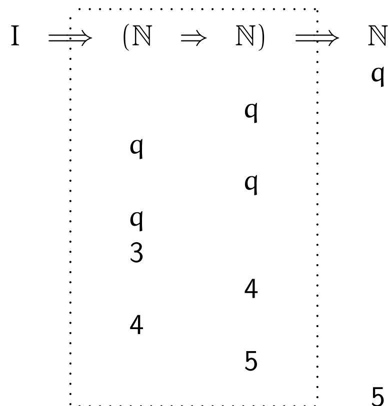
Promoting strategies

To perform this composition, we must say how the successor strategy responds to repeated, nested use.

To do this we construct a repeatable version of the strategy using an operation called *promotion* (cf. linear logic).

The promoted version of the strategy contains all legal interleavings of ordinary plays from the original strategy (roughly).

A picture of the play



Composition = substitution

As usual for a categorical semantics, *composition* in the category models *substitution* in the syntax.
Here we see that the semantics of

$$\mathrm{f}(\mathrm{f}(3))[\mathsf{succ}/\mathrm{f}]$$

is the same as the semantics of the program

$$5.$$

I hope this is not a surprise! ($3 + 1 + 1 = 5$).

Identity

A very important kind of strategy is the *copycat* strategy.

In a game of the form $A \Rightarrow A$, a copycat strategy works by copying O's moves: what O plays in one of the As, P plays in the other.

$$\begin{array}{ccc} A & \Rightarrow & A \\ & & a_1 \\ a_1 & & \\ a_2 & & \\ & & a_2 \\ & \vdots & \end{array}$$

This is an identity for composition.

The expressive power of games

We now have enough structure to model functional programming languages.

However, the semantic universe of games goes beyond the functional world:

- some programs which are equivalent in pure functional languages are distinguished by the model
- many strategies which cannot be defined by pure functional programs are available.

Hyland and Ong wanted a *fully abstract model of PCF*, so they constrained the games model to get rid of non-functional behaviour.

We will embrace the bad behaviour, and end up with a very simple model of a more expressive language.

Intensionality of games

Strategies carry *intensional* (algorithmic) information which cannot be detected in the purely functional world.

$$
\begin{array}{cccc}
x : & \mathbb{B} & \vdash & \mathsf{if}\ x\ \mathsf{then}\ x\ \mathsf{else}\ x : \quad \mathbb{B} \\
 & & & q \\
 & q & & \\
 & b_1 & & \\
 & q & & \\
 & b_2 & & \\
 & & & b_2
\end{array}
$$

This is not the same as $x : B \vdash x : B$, which is a copycat.

In an imperative language where x may be bound to a side-effecting expression, this difference is vital.

History-sensitivity = memory

Consider a strategy with the following two forms of play.

(i)	$(\mathbb{N}$	$\Rightarrow$	$\mathbb{N})$	$\Rightarrow$	$\mathbb{N}$	(ii)	$(\mathbb{N}$	$\Rightarrow$	$\mathbb{N})$	$\Rightarrow$	$\mathbb{N}$
					q						q
			q						q		
	q								n		
	m										$2n+1$
			n								
					$2n$						

Behaviour like this can only be programmed using *mutable store*.

It turns out that considering a programming language which combines functional and imperative programming gives rise to a very simple, very accurate games model.

Idealized Algol

Idealized Algol (IA) is Reynolds's theoretical distillation of Algol 60. IA can be seen as:

- a basic functional language extended with state
  - $x := 3$
  - $!x$
  - new x in ...
- alternatively, a basic imperative language extended with
  - block structure (new)
  - higher-order procedures (λ-calculus).

Either way, it is very expressive, elegant and powerful.

Capturing programming intuition

Programmers in Algol-like languages have many intuitions about the behaviour of programs.

- The use of local variables is invisible from outside a block: privacy, modularity, representation independence.
- State changes are irreversible: there is no “snapback” construct

  $$\mathsf{snapback}(P)$$

  which runs P and then undoes all its state-changes.

A good semantics should capture and formalize these intuitions, and the program optimizations which they justify.

Some program equivalences

Garbage collection If x does not occur free in P then

$$\text{new } x \text{ in } P \cong P.$$

No snapback

$$\text{new } x \text{ in } P(x := 1); \text{if } !x = 1 \text{ then } \Omega \text{ else skip} \cong P(\Omega)$$

Representation Independence

$$\begin{aligned} & \text{new } x : \text{bool in } x := \text{true}; P(!x, x := \neg !x) \\ \cong \quad & \text{new } x : \text{int in } x := 1; P(!x > 0, x := -!x) \end{aligned}$$

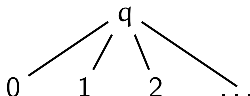
Syntax of IA

The language we will consider is a simply-typed λ-calculus with the following base types.

- Expression types $\mathbb{N}$, $\mathbb{B}$, with the usual constants and operations, including dereferencing of storage variables $!x$.
- The type comm of *commands*. Commands include assignment $x := M$, sequential composition $C_1; C_2$, skip, and local blocks $\mathsf{new}\ x\ \mathsf{in}\ M$.
- The type var of storage variables. Such variables are allocated using new and manipulated via assignment and dereferencing.

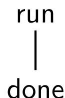
Semantics of base types

The expression types are interpreted as usual. For the natural numbers:



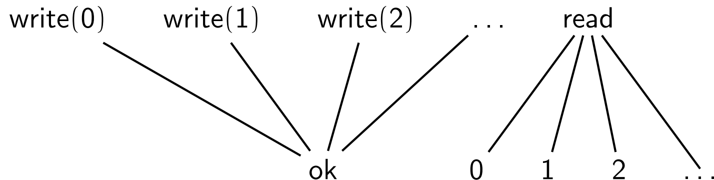
(but note we are going to restrict to a finite set of values very soon.)

Commands are interpreted with a similar game:



Semantics of var

Variables have two kinds of initial move, for reading and for writing:



Some examples

$$
\begin{array}{cccc}
x : \mathsf{var}, & y : \mathsf{var} & \vdash \; x :=\, !y + 1 : & \mathsf{comm} \\
& & & \mathsf{run} \\
& \mathsf{read} & & \\
& n & & \\
\mathsf{write}(n+1) & & & \\
\mathsf{ok} & & & \\
& & & \mathsf{done}
\end{array}
$$

Some examples

$$c : \mathsf{comm}, \quad x : \mathsf{var} \quad \vdash \quad x := 3; c; x := !x + 1 : \quad \mathsf{comm}$$

$c : \mathsf{comm},$	$x : \mathsf{var}$	$\vdash$ $x := 3; c; x := !x + 1 :$	comm
			run
	write(3)		
	ok		
run			
done			
	read		
	n	(n need not be 3)	
	write$(n + 1)$		
	ok		
			done

Bad variable behaviour

In the previous example, O *must* be allowed to respond to read with any value, since the command c may later be bound to something like $x := 19$.

If we wrap the command in a $\mathsf{new}\ x\ \mathsf{in}\ \ldots$, this changes: the command c cannot now access x, and nor can anything else.

The variable x is now a *good variable*.

Variable allocation

The command $\mathsf{new}\ x\ \mathsf{in}\ P$ is just like P, except that:

- x is bound to a storage cell, so P's interactions with x have the expected causal relationship between values written and values read.
- The outside world can no longer see x.

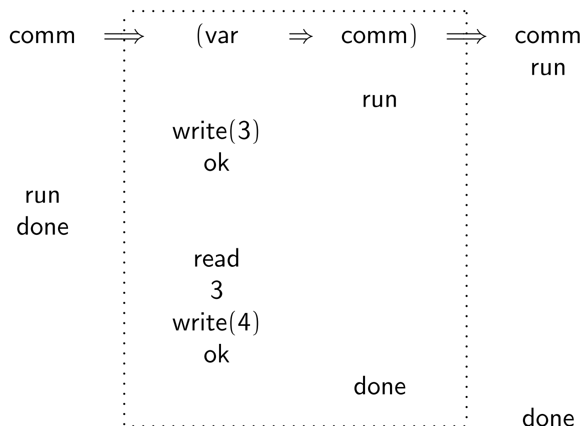
Semantics of allocation

In game semantics, variable allocation is handled by composition with this strategy:

$$
\begin{array}{ccccc}
(\mathsf{var} & \Rightarrow & \mathsf{comm}) & \Rightarrow & \mathsf{comm} \\
 & & & & \mathsf{run} \\
 & & \mathsf{run} & & \\
s & & & & \\
 & & \mathsf{done} & & \\
 & & & & \mathsf{done}
\end{array}
$$

where s is any sequence of reads and writes for which the values read match the last values written at all times.

Allocation in action



Allocation in action

This composition achieves two things:

- good variable behaviour is enforced.
- the interactions in the var type are hidden.

A restricted IA

We will now focus on a restricted fragment of this language and its model, in order to obtain a decidability result.

We consider only terms of the form

$$x_1 : \Theta_1, \ldots, x_n : \Theta_n \vdash M : B$$

where B is a base type and the Θ_i are first-order types.

We do not handle general recursion, but we do include while-loops.

We assume a finite set of data-values.

For convenience, we assume all terms are β-normal, so there are no λ-abstractions, and the only application terms we consider are those of the form

$$xM_1 \ldots M_n.$$

A theorem

For any program M in this language, the games model gives us a strategy $[\![M]\!]$.

A strategy is a set of sequences over a finite alphabet, that is, a language.

It turns out that for this fragment, the strategy $[\![M]\!]$ is a regular language.

Therefore, program equivalence is decidable.

Making game semantics tractable?

To establish the theorem, we describe the game semantics of this fragment using a mildly extended syntax of regular expressions.

This not only gives us the decidability result, but also provides a syntax for manipulating the model, which allows us to write concise proofs of program equivalences.

Extended regular expressions

We use an extended syntax of regular expressions, as follows.

Constants a (singleton), ϵ (empty string), $\bot$ (empty language).

Standard operations $R + S$ (union), $R \cdot S$ (concatenation), R^* (repetition).

Intersection $R \cap S$.

Hiding $R \setminus \alpha$: delete all symbols in the set α.

Any such expression describes a regular language.

Encoding of plays

We must make the disjoint union operation from game semantics explicit. We annotate moves as follows:

$x:\mathbb{N},$	$f:$ $\mathbb{N}$	$\Rightarrow$	$\mathbb{N}$	$\vdash$	$f(x):$ $\mathbb{N}$
					q
			q_f		
	q_f^1				
q_x					
n_x					
	n_f^1				
			m_f		
					m

Denotation of terms

The strategy for a term

$$x_1 : \Theta_1, \ldots, x_n : \Theta_n \vdash M : \mathbb{N}$$

consists of a set of sequences of the form

$$q \ldots n$$

We will give a set of regular languages $(\![M]\!)_n$ such that

$$[\![M]\!] = \sum_n q \cdot (\![M]\!)_n \cdot n$$

Denotation of terms

For M : comm,

$$[\![M]\!] = \mathsf{run} \cdot (\![M]\!) \cdot \mathsf{done}.$$

For M : var,

$$\begin{aligned}[\![M]\!] &= \sum_n \mathsf{read} \cdot (\![M]\!)_{\mathsf{read}(n)} \cdot n \\ &+ \sum_n \mathsf{write}(n) \cdot (\![M]\!)_{\mathsf{write}(n)} \cdot \mathsf{ok}\end{aligned}$$

Constants

In the usual game semantics, $[\![n]\!] = \{q \cdot n\}$.
We define

$$(\![n]\!)_n = \epsilon, \qquad (\![n]\!)_m = \bot \text{ for } m \neq n.$$

Similarly:

$$(\![\mathsf{skip}]\!) = \epsilon, \qquad (\![\Omega]\!) = \bot.$$

Variables

For variables $x : \mathbb{N}$, the semantics is the copycat strategy

$$\begin{array}{ccc} x : \mathbb{N} & \vdash & x : \mathbb{N} \\ & & q \\ q_x & & \\ n_x & & \\ & & n \end{array}$$

so we define

$$(\![x : \mathbb{N}]\!)_n = q_x \cdot n_x .$$

More semantic definitions

$$
\begin{aligned}
[\![C; C']\!] &= [\![C]\!] \cdot [\![C']\!] \\
[\![M + M']\!]_n &= \sum_{m+m'=n} [\![M]\!]_m \cdot [\![M']\!]_{m'}. \\
[\![\textsf{if } B \textsf{ then } C \textsf{ else } C']\!] &= [\![B]\!]_{\textsf{true}} \cdot [\![C]\!] + [\![B]\!]_{\textsf{false}} \cdot [\![C']\!] \\
[\![\textsf{while } B \textsf{ do } C]\!] &= (\,[\![B]\!]_{\textsf{true}} \cdot [\![C]\!]\,)^* \cdot [\![B]\!]_{\textsf{false}} \\
[\![V := M]\!] &= \sum_n [\![M]\!]_n \cdot [\![V]\!]_{\textsf{write}(n)} \\
[\![!V]\!]_n &= [\![V]\!]_{\textsf{read}(n)}
\end{aligned}
$$

Semantics of Application

Given $x : \mathsf{comm} \Rightarrow \mathsf{comm} \Rightarrow \mathsf{comm}$, $M_1, M_2 : \mathsf{comm}$, we need to define $[\![xM_1M_2]\!]$.

Recall that in the semantics of x, after P plays run_x, O may respond in three different ways:

- O can call the first argument; when this argument is bound to M_1, this will lead to $[\![M_1]\!]$ being played out
- O can call the second argument; this will lead to $[\![M_2]\!]$ being played
- O can give the answer done_x, after which the whole program ends with done.

Semantics of Application

We therefore define

$$(\![xM_1M_2]\!) = \mathsf{run}_x \cdot (\mathsf{run}_x^1 \cdot (\![M_1]\!) \cdot \mathsf{done}_x^1 + \mathsf{run}_x^2 \cdot (\![M_2]\!) \cdot \mathsf{done}_x^2)^* \cdot \mathsf{done}_x.$$

Note that each argument may be called many times, or not at all, in any order. This continues until done_x is played.

Semantics of allocation

Let X denote the set of symbols tagged with an x. Let Y be the rest of the alphabet.
The regular language

$$G = Y^* \cdot (\mathsf{read}_x \cdot 0_x \cdot Y^*)^* \cdot (\sum_n \mathsf{write}(n)_x \cdot \mathsf{ok}_x \cdot Y^* \cdot (\mathsf{read}_x \cdot n_x \cdot Y^*)^*)^*$$

describes all those strings in which x has good-variable behaviour.
Now we can define

$$(\![\mathsf{new}\ x\ \mathsf{in}\ P]\!) = ((\![P]\!) \cap G) \setminus X.$$

Full abstraction theorem

Theorem

$$P \cong Q \Leftrightarrow [\![P]\!] = [\![Q]\!] \Leftrightarrow (\![P]\!) = (\![Q]\!).$$

This follows from the original full abstraction theorem for the games model of Idealized Algol.

A simple example

$$
\begin{aligned}
(\![\text{while true do } C]\!) &= (\,(\![\text{true}]\!)_{\mathsf{true}} \cdot (\![C]\!)\,)^* \cdot (\![\text{true}]\!)_{\mathsf{false}} \\
&= (\,\epsilon \cdot (\![C]\!)\,)^* \cdot \bot \\
&= \bot \\
&= (\![\Omega]\!).
\end{aligned}
$$

So while true do $C \cong \Omega$.

Garbage collection

Let M be a program in which x does not occur free.

$$\begin{aligned} (\![\text{new } x \text{ in } M]\!) &= (\,(\![M]\!) \cap G\,) \setminus X \\ &= (\![M]\!) \end{aligned}$$

since x is not free in M so no move of $(\![M]\!)$ is tagged with an x.

No snapback

Let M be $P(x := 1)$; if $!x = 1$ then Ω else skip.
We will show that new x in $M \cong P(\Omega)$.

$$
\begin{aligned}
[\![x := 1]\!] &= \mathsf{write}(1)_x \cdot \mathsf{ok}_x \\
[\![P(x := 1)]\!] &= \mathsf{run}_P \cdot \\
&\quad (\mathsf{run}_P^1 \cdot \mathsf{write}(1)_x \cdot \mathsf{ok}_x \cdot \mathsf{done}_P^1)^* \cdot \\
&\quad \mathsf{done}_P \\
[\![!x = 1]\!]_{\mathsf{true}} &= \mathsf{read}_x \cdot 1_x \\
[\![!x = 1]\!]_{\mathsf{false}} &= \sum_{n \neq 1} \mathsf{read}_x \cdot n_x
\end{aligned}
$$

No snapback, continued

$$
\begin{aligned}
[\![\text{if } !x = 1 \text{ then } \Omega \text{ else skip}]\!] &= [\![!x = 1]\!]_{\text{true}} \cdot [\![\Omega]\!] \\
&+ [\![!x = 1]\!]_{\text{false}} \cdot [\![\text{skip}]\!] \\
&= [\![!x = 1]\!]_{\text{true}} \cdot \bot + [\![!x = 1]\!]_{\text{false}} \cdot \epsilon \\
&= [\![!x = 1]\!]_{\text{false}} \\
&= \sum_{n \neq 1} \text{read}_x \cdot n_x
\end{aligned}
$$

Therefore

$$
[\![M]\!] = \sum_{n \neq 1} \text{run}_P \cdot (\text{run}_P^1 \cdot \text{write}(1)_x \cdot \text{ok}_x \cdot \text{done}_P^1)^* \cdot \text{done}_P \cdot \text{read}_x \cdot n_x
$$

No snapback, concluded

$$[\![M]\!] = \sum_{n \neq 1} \mathsf{run}_P \cdot (\mathsf{run}_P^1 \cdot \mathsf{write}(1)_x \cdot \mathsf{ok}_x \cdot \mathsf{done}_P^1)^* \cdot \mathsf{done}_P \cdot \mathsf{read}_x \cdot n_x$$

We therefore have

$$\begin{aligned} [\![M]\!] \cap G &= \mathsf{run}_P \cdot \mathsf{done}_P \cdot \mathsf{read}_x \cdot 0_x \\ [\![\mathsf{new}\ x\ \mathsf{in}\ M]\!] &= ([\![M]\!] \cap G) \setminus X = \mathsf{run}_P \cdot \mathsf{done}_P. \end{aligned}$$

On the other hand, $[\![\Omega]\!] = \bot$ so

$$\begin{aligned} [\![P(\Omega)]\!] &= \mathsf{run}_P \cdot (\mathsf{run}_P^1 \cdot \bot \cdot \mathsf{done}_P^1)^* \cdot \mathsf{done}_P \\ &= \mathsf{run}_P \cdot \mathsf{done}_P. \end{aligned}$$

Further results

Having established decidability of this fragment, there are some obvious further questions:

- what is the complexity of program equivalence?
- what other fragments are decidable and what are their complexities?

Beyond regular 1: recursion

Adding recursion (as opposed to just iteration) to the language takes us out of the realm of regular languages.

Recursion = fixed point = the "infinite nesting" $\mathtt{f} \vdash \mathtt{f}(\mathtt{f}(\mathtt{f}(\ldots)))$.

This arbitrary nesting of function calls is interpreted by arbitrarily deep nesting of O-P move pairs, leading to strategies which are not regular languages.

Recursion is not regular

$f :$ comm	$\rightarrow$	comm	$\vdash f(f(f(\ldots))) :$ comm
			run
		run	
run			
		run	
run			
	$\vdots$		
		run	
		done	
	$\vdots$		
done			
		done	
done			
		done	
			done

Beyond regular 2: higher order

A similar problem comes up if we include second-order free variables:

$$f : (\mathsf{comm} \rightarrow \mathsf{comm}) \rightarrow \mathsf{comm} \vdash f(\lambda x.x) : \mathsf{comm}$$

run

run

run

run

⋮

Now Opponent can nest calls to f's argument arbitrarily deeply.

Pushdown automata

The languages corresponding to these larger fragments turn out to be those expressible by certain kinds of pushdown automata:

- visibly pushdown automata for the higher-order fragment
- deterministic pushdown automata for recursion.

These discoveries were made by Murawski, Ong and Walukiewicz, who established a complete characterization of the (sensible-looking) decidable fragments of the language.

Characterization of fragments

	Iteration	Ground recursion	Higher recursion
Order 2	PSPACE	DPDA	—
Order 3	EXPTIME	DPDA	UND
Order 4	UND	UND	UND

Implementation

These semantic results provide us with a direct route to automated verification: implement the game semantics and use the decision procedure to check program equivalence.

Ghica, Bakewell and Dimovski have developed two tools:

GameChecker The first implementation, which generates a CSP process corresponding to the game semantics of a program and checks it for safety properties.

Mage A more sophisticated tool which uses lazy, on-the-fly model construction.

Conclusions

- Game semantics for imperative programming languages can be relatively simple!
- A syntax of regular expressions allows us to work with the model by hand.
- Accurate, flexible models like this allow us to establish decidability and complexity results for program equivalence problems.
- Game semantics can be used as a basis for software model checking.