



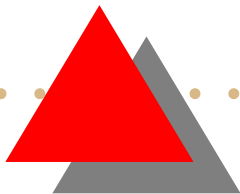
# *Game Semantics: an Overview*

Guy McCusker



## *What did he say?*

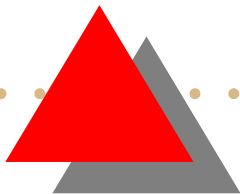
- Qu'est-ce que c'est la sémantique des jeux?  
Or in English...





## *What did he say?*

- Qu'est-ce que c'est la sémantique des jeux?  
Or in English...
- What is it that it is the semantic of games?





# Game Semantics

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

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



## *Prehistory of Game Semantics*

The earliest precursor of game semantics is in the games-based interpretations of logic studied by Lorenzen, Lorenz et al. [6, 10, 11].

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

For example, consider

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





## A dialogue

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

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

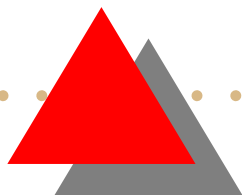
false.

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

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

true.

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



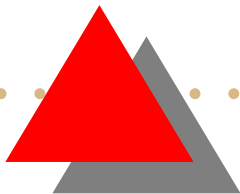


## *Sequential algorithms*

The sequential algorithms models [4] are the first use of the interactive approach in denotational semantics of programs.

A sequential algorithm describes the response of the program to partial input data from the environment.

This was the first model to capture precisely a notion of *sequential, higher-order function*.





## Object Spaces

Reddy's *object spaces* model [13] is a trace-based model of computation, based on intuition of interacting with objects.

For example, a “counter” object might have traces like

$$\text{inc} \cdot \text{inc} \cdot \text{read}(2) \cdot \text{inc} \cdot \text{inc} \cdot \text{read}(4) \cdot \dots$$

Reddy uses this idea to build a model of “interference-free algol”, an important precursor of the games model of Idealized Algol which we will see in the next lecture.

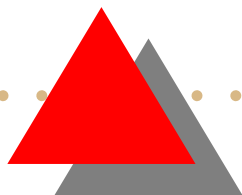




## *Game semantics: the state of the art*

Games models have been built for higher-order programming languages with a variety of computational features:

- pure functional languages (PCF) [1, 8, 12]
- mutable store (Idealized Algol) [3]
- control operators (SPCF, exceptions) [9]
- higher-order store (pointers) [2]
- nondeterminism [7]





## *Full abstraction*

In many cases, the models completely characterize those behaviours which can be programmed in the language: *definability* results hold.

Whenever a definability result holds, it is possible to achieve *full abstraction*: after a straightforward quotient, equivalence in the model coincides with equivalence in the language.

In most cases there is no need for any quotient, if the models are built carefully in the first place.



# 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.



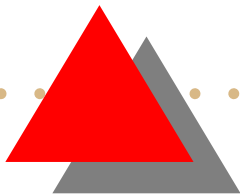


## *Example: a first-order function*

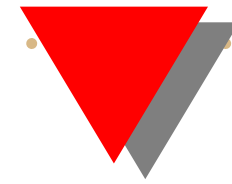
In a first-order function, the system *consumes* input and *produces* output, while the environment *produces* input and *consumes* output.

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.

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

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



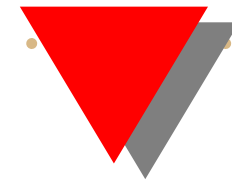
# Higher-Order Functions

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  $f$ ?

# Higher-Order Functions



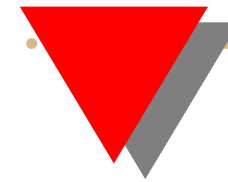
The same ideas extend to model higher-order programs.

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

- 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.



# Higher-Order Functions



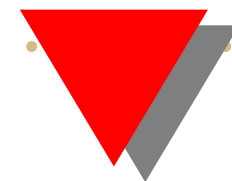
The same ideas extend to model higher-order programs.

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

- 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.

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

- 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.
- 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:

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

$q$   
 $q$   
 $3$   
 $n$   
 $q$   
 $q$   
 $0$   
 $m$

$$m + n$$

# Nested use of arguments

Programs may nest uses of their arguments:

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

$q$

$q$

$q$

$q$

3

$n$

$n$

$m$

$m$



## *Not so simple...*

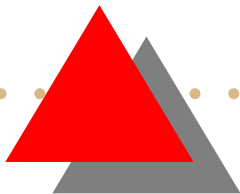
We actually need more data than just the moves that are played.

Consider the terms

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

and

$$f(\lambda x : \mathbb{N}. f(\lambda y : \mathbb{N}. x))$$

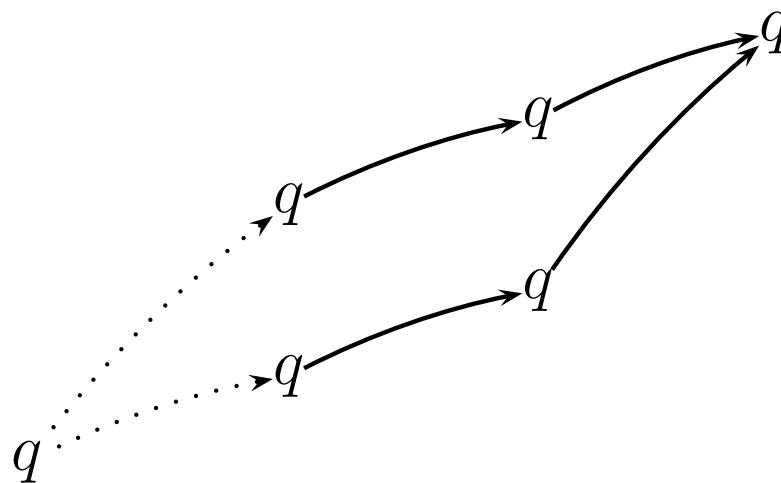




## *Different terms, same plays*

The ambiguity can be resolved by augmenting moves with *justification pointers*:

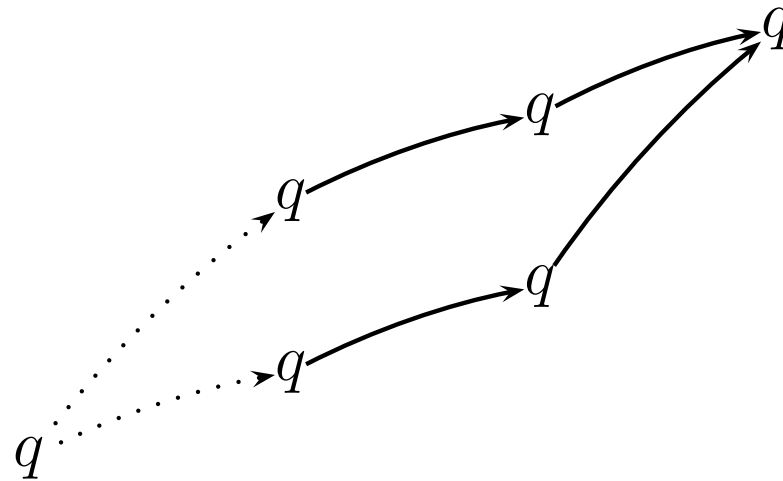
$$((\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$$



## *Different terms, same plays*

The ambiguity can be resolved by augmenting moves with *justification pointers*:

$$((\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$$



But we will ignore the pointers in these lectures!



## *Slightly more formally...*

A *game* is

- a set of *moves*  $M$
- a *labelling function*  $\lambda : M \rightarrow \{O, P\}$
- 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.





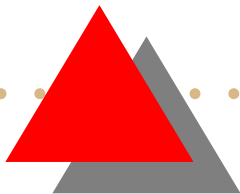
## *Types denote games*

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  $\sigma$  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
- $saa' \in \sigma \Rightarrow s \in \sigma$ .

More constraints will be applied later.





## *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.

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

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.

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

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.

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

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

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

$$\mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{N}$$

$q$

$q$

3

$q$

4

7

# Currying

Compare the previous play to the curried version:

$$\begin{array}{ccc} \mathbb{N} & \Rightarrow & (\mathbb{N} \Rightarrow \mathbb{N}) \\ & & q \\ q & & \\ 3 & & \\ & & q \\ & & 4 \\ & & 7 \end{array}$$

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



## *A category of games*

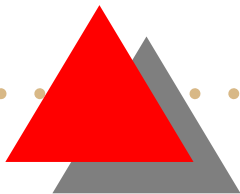
We can now build a category:

Objects:            games

Maps  $A \rightarrow B$ : strategies for  $A \Rightarrow B$

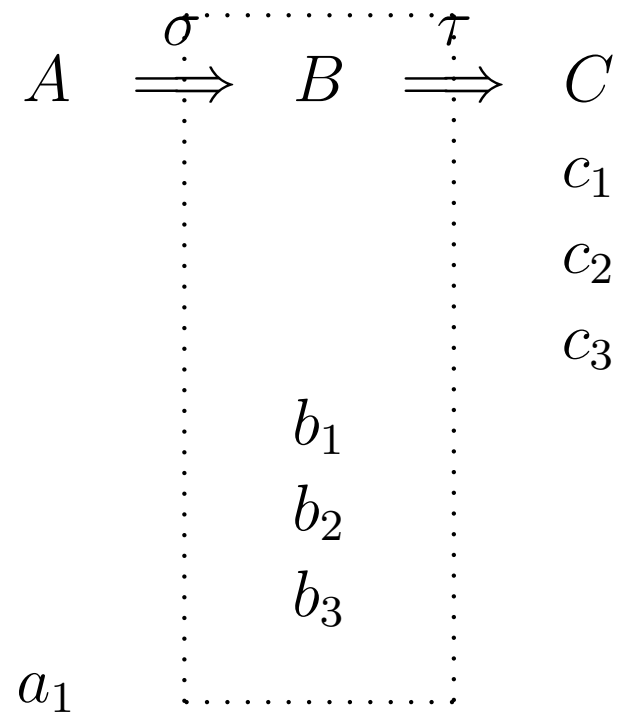
This will turn out to be a cartesian closed category, hence a model of  $\lambda$ -calculus.

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

$$\begin{array}{ccc} 1 & \rightarrow & (\mathbb{N} \Rightarrow \mathbb{N}) \\ & & q \\ & & q \\ & & n \\ & & (n + 1) \end{array}$$

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

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

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

$q$

$q$

$q$

$q$

$q$

$3$

$n$

$n$

$m$

$m$

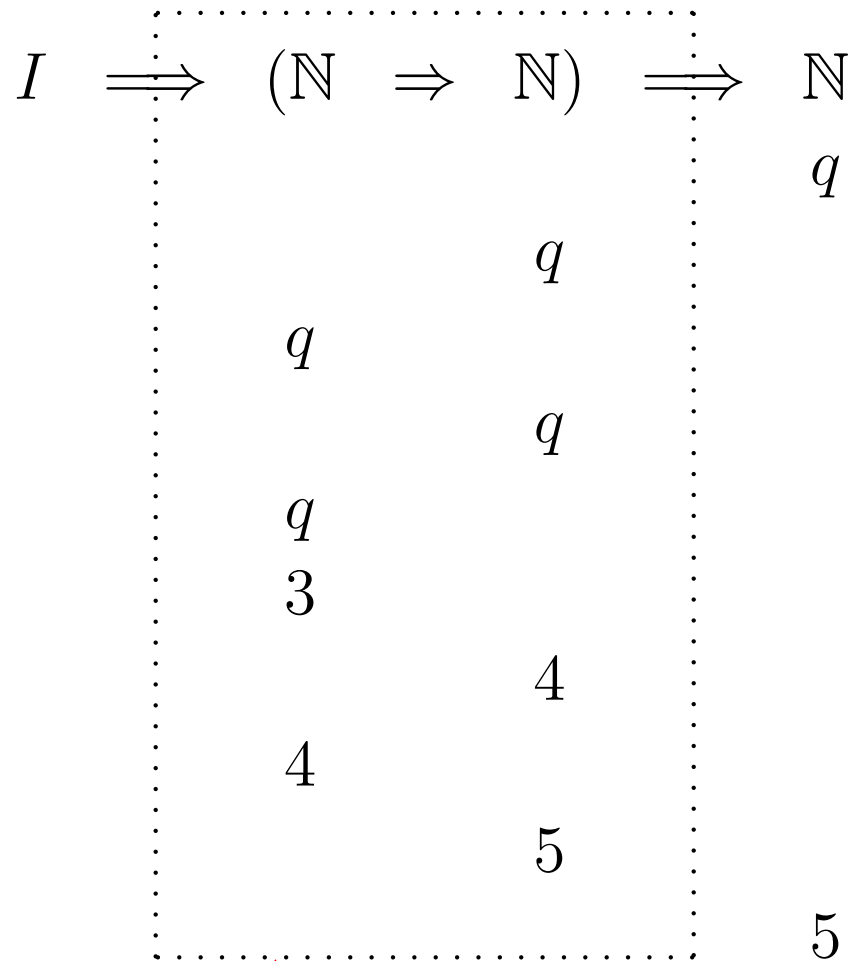



## *Promoting strategies*

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

$$f(f(3))[succ/f]$$

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  $A$ s, P plays in the other.

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

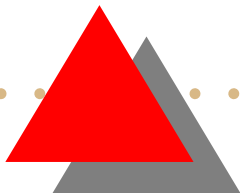
This is an identity for composition.



# *The expressive power of games*

We can now model functional programming languages.  
But the semantic universe of games goes beyond the functional world:

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



## Intensionality of games

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

$$\begin{array}{ccc} x : \mathbb{B} & \vdash & \text{if } x \text{ then } x \text{ else } x : \mathbb{B} \\ & & q \\ & & q \\ & & b_1 \\ & & q \\ & & b_2 \\ & & b_2 \end{array}$$

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



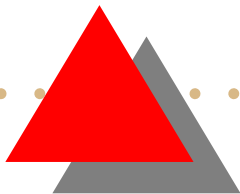
## *Expressivity: nondeterminism*

Strategies can be nondeterministic.

The set

$$\{q \cdot 0, q \cdot 1\}$$

is a valid strategy for  $\mathbb{N}$ .



## Expressivity: catch

Consider a strategy with the following two forms of play.

$$\begin{array}{ccc} (i) & (\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N} & (ii) & (\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N} \\ & q & & q \\ & q & & q \\ q & & n & \\ & 0 & & n + 1 \end{array}$$

This is a simple example of a *control operator*, like the catch construct found in sequential algorithms models [5].

## Expressivity: memory

Consider a strategy with the following two forms of play.

$$\begin{array}{ccc} (i) & (\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N} & (ii) & (\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N} \\ & q & & q \\ & q & & q \\ & q & & n \\ m & & & 2n + 1 \\ & n & & \\ & 2n & & \end{array}$$

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



## *Behavioural constraints*

A key technique in game semantics is to reduce the space of strategies by imposing additional rules. For example, *determinacy* can be enforced by the following constraint on strategies:

$$sab \in \sigma, sac \in \sigma \Rightarrow b = c.$$

We can build a subcategory of deterministic strategies which does not admit our nondeterministic example.



## Constraints: bracketing

We can eliminate control operators like catch as follows.

- classify moves as *questions* and *answers* (demand vs supply of data)
- insist that demand and supply moves match up like properly nested brackets. catch now breaks the rules.

$$\begin{array}{ccccc} \mathbb{N} & \Rightarrow & \mathbb{N} & \Rightarrow & \mathbb{N} \\ & & & & q \\ & & & & q \\ & & q & & \\ & & & & 0 \end{array}$$

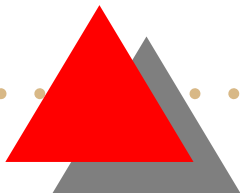


# Classifying programming features

Strategies can be classified according to which behavioural constraints they satisfy.

In the original Hyland-Ong paper on games for PCF, four constraints were identified:

- determinacy
- bracketing
- innocence (a strategy cannot remember the whole history, just a subsequence called the *view*)
- visibility (technical)



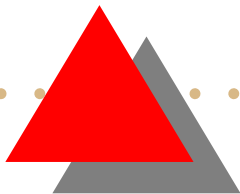


## *A remarkable discovery*

Later work showed that these four constraints each correspond *precisely* to the presence/absence of a certain programming language feature:

determinacy	$\leftrightarrow$	nondeterministic choice
bracketing	$\leftrightarrow$	control operators
innocence	$\leftrightarrow$	mutable store
visibility	$\leftrightarrow$	higher-order store (pointers)

The fully abstract model of PCF has all these ruled. By relaxing them one by one, we obtain fully abstract models of PCF augmented with these programming features, in various combinations.



# References

- [1] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Information and Computation*, 162(2):409–470, 2000.
- [2] Samson Abramsky, Kohei Honda, and Guy McCusker. A fully abstract game semantics for general references. In *Proceedings, Thirteenth Annual IEEE Symposium on Logic in Computer Science*, pages 334–344. IEEE Computer Society Press, 1998.
- [3] Samson Abramsky and Guy McCusker. Full abstraction for Idealized Algol with passive expressions. *Theoretical Computer Science*, 227:3–42, 1999.
- [4] Gérard Berry and P.-L. Curien. Sequential algorithms on concrete data structures. *Theoretical Computer Science*, 20:265–321, 1982.
- [5] R. Cartwright, P.-L. Curien, and M. Felleisen. Fully abstract semantics for observably sequential languages. *Information and Computation*, 111(2):297–401, 1994.
- [6] Walter Felscher. Dialogues as a foundation for intuitionistic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic, Volume III*, pages 341–372. D. Reidel Publishing Company, 1986.

- [7] Russell Harmer and Guy McCusker. A fully abstract game semantics for finite nondeterminism. In *Proceedings, Fourteenth Annual IEEE Symposium on Logic in Computer Science*, pages 422–430, 1999.
- [8] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II and III. *Information and Computation*, 162(2):285–408, 2000.
- [9] Jim Laird. Full abstraction for functional languages with control. In *Proceedings, Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 58–67. IEEE Computer Society Press, 1997.
- [10] P. Lorenzen. Logik und agon. In *Atti del Congresso Internazionale di Filosofia*, pages 187–194, Firenze, 1960. Sansoni.
- [11] P. Lorenzen. Ein dialogisches Konstruktivitätskriterium. In *Infinistic Methods*, pages 193–200, Warszawa, 1961. PWN. Proceed. Symp. Foundations of Math.
- [12] H. Nickau. Hereditarily sequential functionals. In *Proceedings of the Symposium on Logical Foundations of Computer Science: Logic at St. Petersburg*, Lecture notes in Computer Science. Springer, 1994.

- [13] Uday S. Reddy. Global state considered unnecessary: Object-based semantics for interference-free imperative programs. *Lisp and Symbolic Computation*, 9(1), 1996.