# A Coalgebraic Approach to Stateful Objects in Game Semantics

John Gowers and James Laird

May 31, 2017

# Semantics of Stateful Programs

In a *purely functional* language, we can represent a function of type $S \to T$ as a function between sets:

$$[\![f]\!] : [\![S]\!] \to [\![T]\!]$$

# Semantics of Stateful Programs

In a *purely functional* language, we can represent a function of type $S \to T$ as a function between sets:

$$[\![f]\!] : [\![S]\!] \to [\![T]\!]$$

In a stateful language, we have to take the program *state* into consideration.

# Semantics of Stateful Programs

In a *purely functional* language, we can represent a function of type $S \to T$ as a function between sets:

$$[\![f]\!] : [\![S]\!] \to [\![T]\!]$$

In a stateful language, we have to take the program *state* into consideration. We represent a stateful function as a function:

$$[\![f]\!] : W \times [\![S]\!] \to W \times [\![T]\!]$$

where $W$ denotes the program state.

# Game Semantics

In Game Semantics, we denote types by games and we denote elements of types by strategies for those games.

# Game Semantics

In Game Semantics, we denote types by games and we denote elements of types by strategies for those games. We start with very simple games and build up more complicated ones using *connectives*

# Game Semantics

In Game Semantics, we denote types by games and we denote elements of types by strategies for those games. We start with very simple games and build up more complicated ones using *connectives* For example, if $X$ is a set of values, we have a game $\mathrm{Read}[X]$ and a game $\mathrm{Write}[X]$:

$$\mathrm{Read}[X] : \ ?x \ (x \in X) \quad \mathrm{Write}[X] : \ x\checkmark \ (x \in X)$$

# Game Semantics

In Game Semantics, we denote types by games and we denote elements of types by strategies for those games. We start with very simple games and build up more complicated ones using *connectives* For example, if $X$ is a set of values, we have a game Read$[X]$ and a game Write$[X]$:

$$\mathrm{Read}[X] : \ ?x \ (x \in X) \quad \mathrm{Write}[X] : \ x\checkmark \ (x \in X)$$

Given games $A, B$, we can form the *compound games* $A \times B$, $A \otimes B$, $A \multimap B$, $A \oslash B$,... in which the games $A$ and $B$ are played in parallel.

# Function types and Exponentials

Games of the form $A \multimap B$ represent functions from $A$ to $B$.

# Function types and Exponentials

Games of the form $A \multimap B$ represent functions from $A$ to $B$.

$$\mathsf{Read}[\mathbb{N}] \quad \multimap \quad \mathsf{Read}[\mathbb{N}]$$

# Function types and Exponentials

Games of the form $A \multimap B$ represent functions from $A$ to $B$.

$$\begin{array}{ccc} \mathsf{Read}[\mathbb{N}] & \multimap & \mathsf{Read}[\mathbb{N}] \\ & & ? \end{array}$$

# Function types and Exponentials

Games of the form $A \multimap B$ represent functions from $A$ to $B$.

$$\begin{array}{ccc} \mathsf{Read}[\mathbb{N}] & \multimap & \mathsf{Read}[\mathbb{N}] \\ & & ? \\ ? & & \end{array}$$

# Function types and Exponentials

Games of the form $A \multimap B$ represent functions from $A$ to $B$.

$$\begin{array}{ccc} \mathsf{Read}[\mathbb{N}] & \multimap & \mathsf{Read}[\mathbb{N}] \\ & & ? \\ ? & & \\ 7 & & \end{array}$$

# Function types and Exponentials

Games of the form $A \multimap B$ represent functions from $A$ to $B$.

$$
\begin{array}{ccc}
\mathsf{Read}[\mathbb{N}] & \multimap & \mathsf{Read}[\mathbb{N}] \\
 & & ? \\
? & & \\
7 & & \\
 & & 49
\end{array}
$$

# Function types and Exponentials

Games of the form $A \multimap B$ represent functions from $A$ to $B$.

$$\begin{array}{ccc}
\mathrm{Read}[\mathbb{N}] & \multimap & \mathrm{Read}[\mathbb{N}] \\
 & & ? \\
? & & \\
7 & & \\
 & & 49
\end{array}$$

But what if the argument is a function?

# Function types and Exponentials

Games of the form $A \multimap B$ represent functions from $A$ to $B$.

$$\begin{array}{ccc} \mathrm{Read}[\mathbb{N}] & \multimap & \mathrm{Read}[\mathbb{N}] \\ & & ? \\ ? & & \\ 7 & & \\ & & 49 \end{array}$$

But what if the argument is a function? For example, take the lambda term

$$\lambda f^{\mathtt{nat} \to \mathtt{nat}}.f(f\ 3) \colon (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat}$$

# Function types and Exponentials

$$\lambda f^{\mathtt{nat}\to\mathtt{nat}}.f(f\ 3)\colon (\mathtt{nat}\to\mathtt{nat})\to\mathtt{nat}$$

# Function types and Exponentials

$$\lambda f^{\mathtt{nat}\to\mathtt{nat}}.f(f\ 3)\colon (\mathtt{nat}\to\mathtt{nat})\to\mathtt{nat}$$

$$(\mathsf{Read}[\mathbb{N}] \quad \multimap \quad \mathsf{Read}[\mathbb{N}]) \quad \multimap \quad \mathsf{Read}[\mathbb{N}]$$

# Function types and Exponentials

$$\lambda f^{\mathtt{nat}\to\mathtt{nat}}.f(f\ 3)\colon (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat}$$

$$(\mathsf{Read}[\mathbb{N}] \quad \multimap \quad \mathsf{Read}[\mathbb{N}]) \quad \multimap \quad \underset{?}{\mathsf{Read}[\mathbb{N}]}$$

# Function types and Exponentials

$$\lambda f^{\mathtt{nat}\to\mathtt{nat}}.f(f\ 3) \colon (\mathtt{nat}\to\mathtt{nat})\to\mathtt{nat}$$

$$(\mathsf{Read}[\mathbb{N}] \quad \multimap \quad \mathsf{Read}[\mathbb{N}]) \quad \multimap \quad \underset{?}{\mathsf{Read}[\mathbb{N}]}$$

$$?$$

# Function types and Exponentials

$$\lambda f^{\mathtt{nat}\to\mathtt{nat}}.f(f\ 3)\colon (\mathtt{nat}\to\mathtt{nat})\to\mathtt{nat}$$

$$(\mathsf{Read}[\mathbb{N}] \quad \multimap \quad \mathsf{Read}[\mathbb{N}]) \quad \multimap \quad \mathsf{Read}[\mathbb{N}]$$

?

?

?

# Function types and Exponentials

$$\lambda f^{\mathtt{nat}\to\mathtt{nat}}.f(f\ 3) \colon (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat}$$

$$(\mathsf{Read}[\mathbb{N}] \quad \multimap \quad \mathsf{Read}[\mathbb{N}]) \quad \multimap \quad \mathsf{Read}[\mathbb{N}]$$

| | | |
|---|---|---|
| | | ? |
| | ? | |
| ? | | |
| 3 | | |

# Function types and Exponentials

$$\lambda f^{\mathtt{nat} \to \mathtt{nat}}. f(f\ 3) \colon (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat}$$

$$\begin{array}{ccccc}
(\mathsf{Read}[\mathbb{N}] & \multimap & \mathsf{Read}[\mathbb{N}]) & \multimap & \mathsf{Read}[\mathbb{N}] \\
 & & & & ? \\
 & & ? & & \\
? & & & & \\
3 & & & & \\
 & & 9 & &
\end{array}$$

# Function types and Exponentials

$$\lambda f^{\mathtt{nat}\to\mathtt{nat}}.f(f\ 3) \colon (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat}$$

$$\begin{array}{ccccc} (\mathsf{Read}[\mathbb{N}] & \multimap & \mathsf{Read}[\mathbb{N}]) & \multimap & \mathsf{Read}[\mathbb{N}] \\ & & & & ? \\ & & ? & & \\ ? & & & & \\ 3 & & & & \\ & & 9 & & \end{array}$$

...but now we are stuck.

# Function types and Exponentials

$$\lambda f^{\mathtt{nat} \to \mathtt{nat}}.f(f\ 3) \colon (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat}$$

$$\begin{array}{ccccc}
(\mathsf{Read}[\mathbb{N}] & \multimap & \mathsf{Read}[\mathbb{N}]) & \multimap & \mathsf{Read}[\mathbb{N}] \\
 & & & & ? \\
 & & ? & & \\
? & & & & \\
3 & & & & \\
 & & 9 & &
\end{array}$$

...but now we are stuck. We introduce the *exponential* $!A \approx A \oslash (A \oslash (A \oslash \cdots$

# Function types and Exponentials

$$\lambda f^{\mathtt{nat}\to\mathtt{nat}}.f(f\ 3)\colon (\mathtt{nat}\to\mathtt{nat})\to\mathtt{nat}$$

$$\begin{array}{ccccc}
(\mathsf{Read}[\mathbb{N}] & \multimap & \mathsf{Read}[\mathbb{N}]) & \multimap & \mathsf{Read}[\mathbb{N}] \\
 & & & & ? \\
 & & ? & & \\
? & & & & \\
3 & & & & \\
 & & 9 & &
\end{array}$$

...but now we are stuck. We introduce the *exponential* $!A \approx A \oslash (A \oslash (A \oslash \cdots$ Now we represent the type of functions from $A$ to $B$ using the game

$$!A \multimap B$$

# A storage cell

Consider the following (Java) class:

```
public class StorageCell implements AbstractCell {
  private int state;
  public int read() { return state; }
  public void write(int x) { state = x; }
}
```

# A storage cell

Consider the following (Java) class:

```
public class StorageCell implements AbstractCell {
  private int state;
  public int read() { return state; }
  public void write(int x) { state = x; }
}
```

We represent the *public interface* of the class with the game

$$!(\mathsf{Write}[\mathtt{int}] \times \mathsf{Read}[\mathtt{int}])$$

# A storage cell

Consider the following (Java) class:

```
public class StorageCell implements AbstractCell {
  private int state;
  public int read() { return state; }
  public void write(int x) { state = x; }
}
```

We represent the *public interface* of the class with the game

$$!(\mathsf{Write}[\mathtt{int}] \times \mathsf{Read}[\mathtt{int}])$$

The *implementation* now corresponds to a *strategy* for that game:

$$7 \checkmark \; ? \; 7 \; ? \; 7 \; 8 \checkmark \; 9 \checkmark \; ? \; 9 \ldots$$

# A state-transformer based approach

# A state-transformer based approach

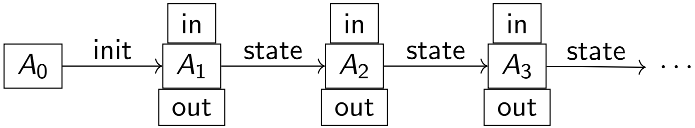
The strategy is easy to define, but it is hard to study because the state is *implicit* or *encapsulated*.

# A state-transformer based approach

The strategy is easy to define, but it is hard to study because the state is *implicit* or *encapsulated*. We want to return to the classical state-transformer approach

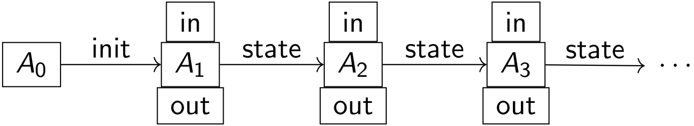
# A state-transformer based approach

The strategy is easy to define, but it is hard to study because the state is *implicit* or *encapsulated*. We want to return to the classical state-transformer approach



# A state-transformer based approach

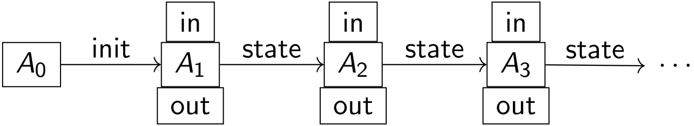
The strategy is easy to define, but it is hard to study because the state is *implicit* or *encapsulated*. We want to return to the classical state-transformer approach




We may construct a state transformer (strategy) $!X \multimap (\mathrm{Read}[X] \times \mathrm{Write}[X]) \oslash !X$ corresponding to a single invocation of the Java class on the previous slide.

# A state-transformer based approach

The strategy is easy to define, but it is hard to study because the state is *implicit* or *encapsulated*. We want to return to the classical state-transformer approach




We may construct a state transformer (strategy) $!X \multimap (\mathrm{Read}[X] \times \mathrm{Write}[X]) \oslash !X$ corresponding to a single invocation of the Java class on the previous slide.

Using some category-theoretic magic ($!A$ is the *final coalgebra* for the endofunctor $A \oslash \_$), we can automatically construct our strategy for $!X \multimap !(\mathrm{Read}[X] \times \mathrm{Write}[X])$ from this state transformer.