Game semantics of references, sequoidally!

Guy McCusker

The logic of higher-order functions

$$
\begin{tikzcd}
\lambda\text{-calculus} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & \text{CCCs} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & \text{intuitionistic logic} \arrow[d, leftrightarrow] \\
\text{linear } \lambda\text{-calculus} \arrow[r, leftrightarrow] & \text{SMCC}, \&, ! \arrow[r, leftrightarrow] & \text{linear logic} \\
 & \text{games} \arrow[u] &
\end{tikzcd}
$$

More exciting things?

A similar theory is coming into place for languages with control, e.g. control categories and the CPS transformation.

Many other kinds of programming language do not admit such an analysis. What can we say about *stateful computation*?

Games models provide (in some sense) a definitive analysis of languages with state. Can we use this as a basis for such an analysis?

Heavy duty game semantics. . .

In these lectures we will see:

- How games can be used to model *pointers* in programming languages.
- How one can prove *definability* and *full abstraction* results for games models.
- How the structure of the games model exposes an underlying *logic of references*.

Maybe.

A quick reminder

Let's briefly review what the game semantics of various programs looks like. A simple example:

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

Multiple use of arguments

Many useful programs use their arguments repeatedly:

$$
\begin{array}{cccccc}
f: & \mathbb{N} & \rightarrow & \mathbb{N} & \vdash f(3) + f(0): & \mathbb{N} \\
 & & & & & q \\
 & & & q & & \\
 & q & & & & \\
 & 3 & & & & \\
 & & & n & & \\
 & & & q & & \\
 & q & & & & \\
 & 0 & & & & \\
 & & & m & & \\
 & & & & & m+n
\end{array}
$$

Employing linear logic

Games naturally model a *linear* type system (actually, affine), where functions use their arguments at most once. To allow multiple use, we introduce the ! connective, and interpret functions $A \Rightarrow B$ using $!A \multimap B$.
A term like $f(f(3))$ is interpreted as a strategy for the game

$$!(!\mathbb{N} \multimap \mathbb{N}) \multimap \mathbb{N}.$$

The game $!A$ consists of countably many copies of A, to be used in order.

Multiple uses via !

$$
\begin{array}{cccccc}
f: & !(!\mathbb{N} & \multimap & \mathbb{N}) & \vdash f(3) + f(0): & \mathbb{N} \\
 & & & & & q \\
 & & & (0,q) & & \\
 & (0,(0,q)) & & & & \\
 & (0,(0,3)) & & & & \\
 & & & (0,n) & & \\
 & & & (1,q) & & \\
 & (1,(0,q)) & & & & \\
 & (1,(0,0)) & & & & \\
 & & & (1,m) & & \\
 & & & & & m+n
\end{array}
$$

Can we simplify !?

This tagging of the separate copies is annoying.
Can we perhaps get away without it? Let's see what happens.

Nested use of arguments

Programs may nest uses of their arguments:

$$
\begin{array}{cccccc}
f : & \mathbb{N} & \Rightarrow & \mathbb{N} & \vdash f(f(3)) : & \mathbb{N} \\
 & & & & & q \\
 & & & q & & \\
 & q & & & & \\
 & & & q & & \\
 & q & & & & \\
 & 3 & & & & \\
 & & & n & & \\
 & n & & & & \\
 & & & m & & \\
 & & & & & m
\end{array}
$$

So far so good…

But consider the terms

$$
\begin{array}{rl}
f : (\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N} \vdash & f(\lambda x : \mathbb{N}. f(\lambda y : \mathbb{N}. y)) \\
\text{and} & f(\lambda x : \mathbb{N}. f(\lambda y : \mathbb{N}. x))
\end{array}
$$

Different terms, same plays

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

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

Trouble!

This is bad news because these two terms do not behave in the same way.

Exercise Find a context of type $\mathbb{N}$ which distinguishes these two programs.

It can be done in a purely functional language, just using arithmetic and conditionals.
It's not easy.
If you allow yourself to use side-effects, it's a lot easier.

So we need tags

The tagging of moves in the ! games is needed precisely to disambiguate these two terms.

$$\lambda f.f(\lambda x.f(\lambda y.?))\lambda f.f(\lambda x.f(\lambda y.x))\lambda f.f(\lambda x.f(\lambda y.y))$$

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

q

$q(0, q)$

$q(0, (0, q))$

$q(1, q)$

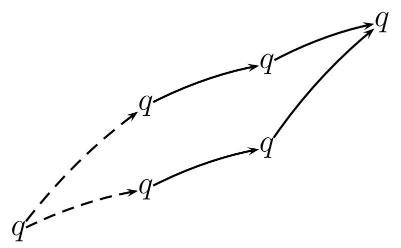
$q(1, (0, q))$

$q(0, (0, (0, q)))(1, (0, (0, q)))$

A different approach

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

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



This is what happens in Hyland-Ong style games.

Games are expressive

Whichever approach we take, we end up with a very revealing semantic universe. Let's look at what kinds of things games models can expose.

Intensionality of games

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

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

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

Expressivity: nondeterminism

Strategies can in general be nondeterministic.
Unless we explicitly rule out nondeterminism, 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.

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

This is a simple example of a *control operator*, like the `catch` construct found in sequential algorithms models [2].

Expressivity: memory

Consider a strategy with the following two forms of play.

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

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 and Abramsky-Jagadeesan-Malacaria papers on games for PCF, several constraints were identified:

- determinacy
- bracketing
- history-freeness (AJM)
- innocence (HO): a strategy cannot remember the whole history, just a subsequence called the *view*, defined using the pointers
- visibility (HO): 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/history-freeness	$\leftrightarrow$	mutable store
visibility	$\leftrightarrow$	higher-order store

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

Analysing constrained strategies

The key to the full abstraction proofs is to show that *definability* results hold:

> every (finite) strategy subject to the appropriate constraints is defined by some term of the language.

A functional example

Consider a strategy on the game $(!A \multimap \mathbb{N}) \multimap \mathbb{N}$ subject to the *bracketing* and *history freeness* conditions.

$$
\begin{array}{ccccc}
(!A & \multimap & \mathbb{N}) & \multimap & \mathbb{N} \\
 & & & & q \\
 & & q & & \\
\vdots & & & & \\
s & & & & \\
\vdots & & & & \\
 & & m & & \\
 & & & & n
\end{array}
$$

Analysis using constraints

- By bracketing, we know that all questions that are asked are answered in order.
- By history-freeness, we know that the final answer n cannot depend on the play s in $!A$, but only on m.
- History-freeness also tells us that the play in $!A$ is not dependent on anything from outside $!A$.

Decomposition

These observations mean that we can *decompose* such a strategy into

- a strategy on $\mathbb{N} \multimap \mathbb{N}$
- a strategy on $!A$.

Further analysis of the history-free strategies shows that any strategy on $!A$ is the promotion of one on A.
Hence any such strategy is of the form

$$f : A \to \mathbb{N} \vdash g(f(a)) : \mathbb{N}$$

where $g : \mathbb{N} \multimap \mathbb{N}$ and $a : A$.

Definability

This decomposition is the key to an inductive proof of definability:

- break a strategy down into smaller pieces
- find terms that define these pieces
- show how to piece them back together to form the original strategy.

Modelling exciting languages

If we impose *none* of these rules, we get

- a very simple-to-define category of games
- enough freedom to model very sophisticated language features
- an apparent lack of structure.

We will shortly see that there is some interesting structure in the unconstrained games model. First let us revisit the semantics of imperative programs.

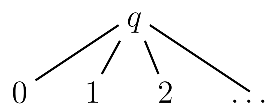
Syntax of IA

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

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

Semantics of base types

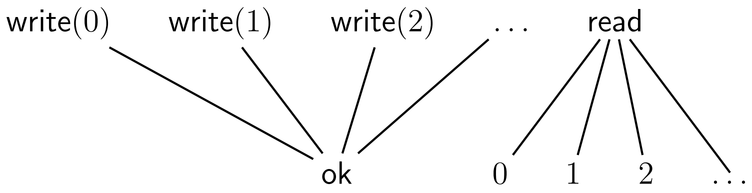
The expression types are interpreted as usual. For the natural numbers:



Commands are interpreted with a similar game:

Semantics of var

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



Some examples

x : var,	y : var	$\vdash$ $x :=!y+1$:	$\mathbb{C}$
			run
	read		
	n		
write$(n+1)$			
ok			
			done

Some examples

$c : \mathbb{C}$,	x : var	$\vdash$ $x := 3; c; x :=!x+1$:	$\mathbb{C}$
			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 new x 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 new x 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 the cell strategy:

cell : I	$\multimap$	!var
		write(3)
		ok
		read
		3
		write(9)
		ok
		read
		9
		$\vdots$

Allocation in action

I	$\multimap$	!var	$\multimap$	$(\mathbb{C}$	$\Rightarrow$	$\mathbb{C})$
						run
		write(3)				
		ok				
				run		
				done		
		read				
		3				
		write(4)				
		ok				
						done

Allocation in action

This composition achieves two things:

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

Notice also that cell does not possess the logical property which partially characterizes functional programming: it is a strategy on a !-type which does not arise from promotion.

Higher-order cells

Can we use similar techniques to model *higher-order store*, i.e. the ability to store functions and other interesting things in variables?

$$x := \lambda y.y + 3$$

How would a cell work in this case?

$$\begin{array}{ccc} \mathsf{cell} : I & \multimap & !\mathsf{var} \\ & & \mathsf{write}(?) \end{array}$$

Isn't it obvious?

Isn't it obvious that it should look like this:

$$\begin{array}{ccc} \mathsf{cell} : I & \longrightarrow & !\mathsf{var} \\ & & \mathsf{write}(\lambda y.y + 3) \\ & & \mathsf{ok} \\ & & \vdots \end{array}$$

NO!

- This is using syntax in the semantics: bad idea.
- It's not at all clear what would happen when we read. Is this even well-defined? (Hint: see the word in red above.)

Another approach

A different interpretation of the var type for storing natural numbers is to set

$$\mathsf{var} = (\mathbb{N} \Rightarrow \mathbb{C}) \times \mathbb{N}.$$

Another approach

With this version, cell plays as follows:

$$\begin{array}{cccccc} (\mathbb{N} & \Rightarrow & \mathbb{C}) & \times & \mathbb{N} \\ & & \mathsf{run} & & \\ q & & & & \\ 3 & & & & \\ & & \mathsf{done} & & \\ & & & & q \\ & & & & 3 \\ & & & \vdots & \end{array}$$

Maybe this will generalise?

A higher-order var type

To store functions of type $\mathbb{N} \Rightarrow \mathbb{N}$, we could try using the type

$$((\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{C}) \times (\mathbb{N} \Rightarrow \mathbb{N}).$$

What should a cell do when O tries to perform the assignment

$$x := \lambda y.y + 3$$

The cell strategy: Second attempt

$$\begin{array}{ccccccccc}
((\mathbb{N} & \Rightarrow & \mathbb{N}) & \Rightarrow & \mathbb{C}) & \times & (\mathbb{N} & \Rightarrow & \mathbb{N}) \\
 & & & & \mathsf{run} & & & & \\
 & & q & & & & & & \\
q & & & & & & & & \\
? & & & & & & & &
\end{array}$$

Oh no! Before the assigment has terminated, we're asking O to tell us what argument to supply to the function. That's no good.

The cell strategy: yet another try

In "real life" (if there is such a thing), assignments terminate immediately:

$$\begin{array}{ccccccccc}
((\mathbb{N} & \Rightarrow & \mathbb{N}) & \Rightarrow & \mathbb{C}) & \times & (\mathbb{N} & \Rightarrow & \mathbb{N}) \\
 & & & & \mathsf{run} & & & & \\
 & & & & \mathsf{done} & & & & \\
 & & & & & & & & q
\end{array}$$

Now what? O is asking us for information about the stored function, but we don't know anything, because we didn't ask.

Final attempt

In this situation, there's only one choice: ask about that function *now*.

$$\begin{array}{ccccccccc}
((\mathbb{N} & \Rightarrow & \mathbb{N}) & \Rightarrow & \mathbb{C}) & \times & (\mathbb{N} & \Rightarrow & \mathbb{N}) \\
 & & & & \mathsf{run} & & & & \\
 & & & & \mathsf{done} & & & & \\
 & & & & & & & & q \\
 & & q & & & & & & \\
q & & & & & & & & \\
 & & & & & & q & &
\end{array}$$

and now play copycat between the reading part and the writing part.

Projection conditions?

Recall that in any play, if we focus our attention on the moves in a subgame, we are supposed to get a proper play of that game. But. . .

	$((\mathbb{N}$	$\Rightarrow$	$\mathbb{N})$	$\Rightarrow$	$\mathbb{C})$	$\times$	$(\mathbb{N}$	$\Rightarrow$	$\mathbb{N})$
O					run				
P					done				
O									q
P			q						
O	q								
P							q		

It's not alternating! Can we avoid this?

A motivating program

Consider this nasty looking program. (It will get nastier.)

$$\lambda f : (\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}. f(\lambda x : \mathbb{N}.3) + 0$$

When we interact with this program, here's what should happen:

- f is called.
- If f calls its argument function, the result 3 is returned to f.
- If f returns a value n, the whole program returns n.

A motivating program

Here's that dialogue in game semantics terms:

$((\mathbb{N}$	$\Rightarrow$	$\mathbb{N})$	$\Rightarrow$	$\mathbb{N})$	$\Rightarrow$	$\mathbb{N}$
						q
				q		
		q				
		3				
				n		
						n

Adding an assignment

Consider this amended program:

$$\lambda f : (\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}. f(\lambda x : \mathbb{N}. v := \lambda y.x; 3) + 0$$

We've added an assignment statement which stores a function in a variable v. Note two points:

- When this assignment happens, the variable x is *not* evaluated.
- The assignment has no influence on the rest of the program.

So if we're not looking at v, for example if v is a hidden, local variable, this program will engage in the same play as the previous one.

Using the variable

Now let's look up the value stored in v;

$$\lambda f : (\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}. f(\lambda x : \mathbb{N}. v := \lambda y. x); 3) + {\color{red}(!v)0}$$

The value stored in v is the constant function $\lambda y.x$. That means that when we come to this part of the term, we need to find out the value of x, i.e. the value f passes to its argument.

The strategy should be:

$$\begin{array}{ccccccc}
((\mathbb{N} & \Rightarrow & \mathbb{N}) & \Rightarrow & \mathbb{N}) & \Rightarrow & \mathbb{N} \\
 & & & & & & q \\
 & & & & q & & \\
 & & q & & & & \\
 & & 3 & & & & \\
 & & & & n & & \\
q & & & & & & \\
m & & & & & & \\
 & & & & & & n+m
\end{array}$$

This violates all sorts of rules of game semantics, but it looks like we need it.

It really works

This looks horrific, but it in fact forms the basis of a (fully abstract) model of references.
The model first appeared in 1996 [1]. We will look at a slightly different model based on the same idea, due to Laird [3].
We need some new definitions of games where projection conditions are not applied. In fact we almost allow any sequence of moves which alternates between O and P.

Any sequence?

There are three reasons not to allow just any old sequence of moves:

- For the additive conjunction $A \& B$, if O moves in A first, we want to prevent any moves being made in B, and vice versa.
- For the exponential $!A$, we want to ensure that the copies are opened in numerical order
- It would be far too easy that way.

Laird games

A game A of the new kind is given by:

- A set M_A of moves.
- A labelling function $\lambda_A : M_A \to \{O, P\}$.
- A set $S_A \subseteq M_A$ of *starting moves* (all belonging to O).
- A relation $\triangleright_A$ between moves: $m \triangleright_A n$ means m *triggers* (makes playable) n.
- A relation $\triangleleft_A$ between moves: $m \triangleleft_A n$ means m *blocks* (makes unplayable) n.

Valid plays

A play of a game is a sequence of moves such that:

- O starts, and O/P alternate after that.
- The first move comes from S_A.
- After the first move, a move can only be played if a previous move triggers it, and no previous move blocks it.

In the games we will use, all moves will be self-blocking, so no move is played more than once.

Connectives

We can interpret $\otimes$, $\multimap$ and $\&$ straightforwardly.

$$\begin{array}{rcl} M_{A\otimes B} & = & M_A + M_B \\ \lambda_{A\otimes B} & = & [\lambda_A, \lambda_B] \\ S_{A\otimes B} & = & S_A + S_B \\ m \triangleright_{A\otimes B} n & \iff & m \triangleright_A n \vee m \triangleright_B n \\ m \triangleleft_{A\otimes B} n & \iff & m \triangleleft_A n \vee m \triangleleft_B n \end{array}$$

The empty game I is the tensor unit as usual.

Implication

$$\begin{array}{rcl} M_{A\multimap B} & = & M_A + M_B \\ \lambda_{A\multimap B} & = & [\overline{\lambda_A}, \lambda_B] \\ S_{A\multimap B} & = & S_B \\ m \triangleright_{A\multimap B} n & \iff & m \triangleright_A n \vee m \triangleright_B n \vee \\ & & (n \in S_A \wedge m \in S_B) \\ m \triangleleft_{A\multimap B} n & \iff & m \triangleleft_A n \vee m \triangleleft_B n \end{array}$$

Conjunction

$$
\begin{array}{rcl}
M_{A\&B} & = & M_A + M_B \\
\lambda_{A\&B} & = & [\lambda_A, \lambda_B] \\
S_{A\&B} & = & S_A + S_B \\
m \rhd_{A\&B} n & \Longleftrightarrow & m \rhd_A n \vee m \rhd_B n \\
m \lhd_{A\&B} n & \Longleftrightarrow & m \lhd_A n \vee m \lhd_B n \vee \\
 & & (m \in S_A \wedge n \in M_B) \vee \\
 & & (m \in S_B \wedge n \in M_A)
\end{array}
$$

A category

We define strategies as usual, and build a category in the standard game-semantical way:

- Objects are games
- Morphisms from A to B are strategies on $A \multimap B$
- Identities are copycat strategies
- Composition is the usual plugging together of strategies.

The structure we've described makes this a symmetric monoidal closed category with products.

A new connective

It will be useful to have a new connective around: the *sequoidal product* $\oslash$.
Think of it as a "sequenced monoid": it's just like $\otimes$ but play has to start on the right.

$$
\begin{array}{rcl}
M_{A\oslash B} & = & M_A + M_B \\
\lambda_{A\oslash B} & = & [\lambda_A, \lambda_B] \\
S_{A\oslash B} & = & S_B \\
m \rhd_{A\oslash B} n & \Longleftrightarrow & m \rhd_A n \vee m \rhd_B n \vee \\
 & & (n \in S_A \wedge m \in S_B) \\
m \lhd_{A\oslash B} n & \Longleftrightarrow & m \lhd_A n \vee m \lhd_B n
\end{array}
$$

Some interesting structure

There are obvious copycat strategies

$$
\begin{array}{l}
\mathsf{l} : A \otimes B \longrightarrow B \oslash A \\
\mathsf{r} : A \otimes B \longrightarrow A \oslash B
\end{array}
$$

and hence a map $A \otimes B \longrightarrow (B \oslash A)\&(A \oslash B)$.
A key point is that this map has an inverse

$$(B \oslash A)\&(A \oslash B) \longrightarrow A \otimes B.$$

This says that in a tensor game, it's possible to play differently depending on where O starts: implicit information flow across the tensor!

Exponential

As before we can define $!A$ as "infinitely many copies of A". Using $\oslash$ we can readily express the condition that the copies are used in a fixed order.

$$!A = \cdots \oslash A \oslash A \oslash A.$$

So $!A$ is a recursively defined type: $!A = !A \oslash A$.
(It is possible to give the definition of $!A$ directly: try it sometime!)

A programming language

Let's see how to model a programming language. It will be based on the λ-calculus, so its types are

$$T ::= \mathbf{0} \mid T \Rightarrow T.$$

Here $\mathbf{0}$ is an empty type. We're not including any datatypes, or very much at all. More can be added, and CPS translations can encode much more.
The terms of the language that interest us are

$$M, N ::= x \mid \lambda x.M \mid MN \mid x := M; N$$

that is: variables, abstraction, application, and an assignment operator.

Typing the assignment operator

In addition to the usual typing rules of the λ-calculus, we have

$$\frac{\Gamma, x : T \vdash M : T \quad \Gamma, x : T \vdash N : S}{\Gamma, x : T \vdash x := M; N : S}$$

This operation is like the `set!` operation in Scheme: it updates the value of x so that future references to x evaluate to M.
It's a very powerful feature and requires a very unusual game semantics, as we have seen.

Modelling the language

Types are interpreted in the usual way: we set

$$[\![T \Rightarrow S]\!] = ![\![T]\!] \multimap [\![S]\!]$$

using the linear decomposition of intuitionistic implication.
The empty type is modelled as a game with just one move.

A bit of encoding

We can encode the booleans in the usual λ-calculus style, using

$$\mathbf{0} \Rightarrow \mathbf{0} \Rightarrow \mathbf{0}.$$

The game interpretation of this is the usual game for the booleans. Similarly, the type of commands is

$$\mathbf{0} \Rightarrow \mathbf{0}.$$

What about terms?

We cannot get away with the usual interpretation of a term

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

as a morphism

$$![\![T_1]\!] \otimes \cdots \otimes ![\![T_k]\!] \to [\![T]\!]$$

because this gives no way to account for the assignment operator: the variables $x_1, \ldots, x_k$ are assignable!

A type of assignable variables

Following our earlier discussion, let us define a new operation on types: $\S A$ is the type of assignable variables which can store entities of type A.
We set

$$\S A = (!A \multimap \mathbb{C}) \& A.$$

We will interpret a term

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

as a morphism

$$!\S[\![T_1]\!] \otimes \cdots \otimes !\S[\![T_k]\!] \to [\![T]\!].$$

Semantics of terms

First the easy bit: since terms of type A are interpreted as maps into A, and $[\![A \Rightarrow B]\!] = ![\![A]\!] \multimap [\![B]\!]$, application is modelled in the completely standard method in a linear category.
That is, to interpret MN we form a tuple of

- the semantics of M, a map into $!A \multimap B$,
- the semantics of N, a map into A, promoted to be a map into $!A$

and compose with the application map from our SMCC structure.

Assignment

The assignment operation is two things in one:

$$x := M; N.$$

First it stores M in x, then it behaves like N.

Storing M in x

To model the storing part, we use the “assignment method” from the variable type: we have a projection

$$\S A \to (!A \multimap \mathbb{C}).$$

Since M is a morphism into A, we model the assignment by applying this function to M.

Carrying on as N

The sequential composition part of the assignment is then modelled using the strategy

$$\begin{array}{ccccc}
\mathbb{C} & \otimes & A & \longrightarrow & A \\
 & & & & a \\
\text{run} & & & & \\
\text{done} & & & & \\
 & & a & &
\end{array}$$

copycat

A logical view of this

For the empty type, this sequential composition really has type

$$\mathbf{0} \otimes (\mathbf{0} \multimap \mathbf{0}) \longrightarrow \mathbf{0}$$

so it is just application!
The same operation at higher types can be built up from this functorially.

Variables

The semantics of a variable is easy: we use a projection to the *lookup* part of the § type

$$[\![\Gamma, x : A \vdash x : A]\!] : !\S\Gamma \otimes !\S A \to A.$$

This is not the usual categorical interpretation of a variable: we have to take account of the fact that our variables are assignable, and the expression x means “look up the value stored in x”.

Abstraction

Abstraction in this language is a little odd. It really consists of allocating a new storage variable, and providing a function which stores its argument in that variable.
A term

$$x : A \vdash M : B$$

gives us a morphism $!\S A \to B$.
We need to turn this into something of type $!A \multimap B$.
Just as with allocation in IA, we do this by plugging in a cell strategy, this time of type

$$!A \to !\S A.$$

The higher-order cell

The strategy looks like this:

$$!A \quad \longrightarrow \quad !(!A \quad \multimap \quad \mathbb{C}) \quad \otimes \quad !A$$

copycat

run

done

copycat

In words…

- At first, if O plays in the rightmost $!A$, we play copycat to the leftmost copy.
- If O plays run in index i of $!(!A \multimap \mathbb{C})$, we respond with done.
- Now when O plays in the rightmost $!A$, we play copycat to the $!A$ in index i of $!(!A \multimap \mathbb{C})$.

So the indices of ! tie up reads to writes.
Projection conditions are spectacularly violated.

It all works

This gives a sound model of our programming language. Proving this is less than straightforward!
Furthermore, the model is *fully abstract*. We'll soon have a look at how this is proved, but first, let's analyse the structure of the model a little.

Strict maps

Suppose A and B are games and $\sigma : A \to B$ is a strategy.
We say σ is *strict* if it responds to any initial move in B with an initial move in A, or has no response.
Strict strategies form a subcategory $\mathcal{G}_s$ of our category $\mathcal{G}$.
That is, identities are strict and strictness is preserved by composition.

$\oslash$ and $\mathcal{G}_s$

The category $\mathcal{G}_s$ lets us state some properties of $\oslash$. First notice that it is a functor

$$\oslash : \mathcal{G} \times \mathcal{G}_s \to \mathcal{G}_s.$$

Now consider strict maps

$$A \oslash B \to C.$$

These must respond to the first move in C by playing in B, so they are the same as strict maps

$$B \to (A \multimap C).$$

Hence $A \oslash -$ is left-adjoint to $A \multimap - : \mathcal{G}_s \to \mathcal{G}_s$.

An isomorphism

Slightly weirder is the fact that

$$A \oslash (B \multimap C) \cong B \multimap (A \oslash C).$$

The unit of the adjunction we just saw gives us a strict map

$$B \longrightarrow (A \multimap A \oslash B)$$

so this isomorphism gives us a strict map

$$B \longrightarrow A \oslash (A \multimap B)$$

In fact this is the unit of another adjunction: $A \oslash -$ is also *right* adjoint to $A \multimap -$.

In technical terms…

There is a technical term for this situation…

AMAZING!

(I believe these things are called *amazing right adjoints*.)

It's actually useful

This gives us a logical way to build up our cell strategy: the key to the cell is the ability to jump back in to the writing part of the play:

$$
\begin{array}{ccccc}
A & \oslash & (A & \multimap & \mathbb{C}) \\
 & & & & \text{run} \\
 & & & & \text{done} \\
a & & & & \\
 & & a & & \\
\end{array}
$$

copycat

Across the isomorphism

On the other side of the isomorphism, this is a far less controversial play:

$$
\begin{array}{ccccc}
A & \multimap & (A & \oslash & \mathbb{C}) \\
 & & & & \text{run} \\
 & & & & \text{done} \\
 & & a & & \\
a & & & & \\
\end{array}
$$

copycat

which is always present for logical reasons.

More consequences

Laird also shows that

- Any category with this structure (an SMCC with $\oslash$ and the two additional adjunctions) can interpret recursion: formally, it is a *traced monoidal category*.
- Any category with this structure plus ! and $\&$ can model the programming language we're considering.
- With a few extra assumptions, the model will be fully abstract.

Decomposition

The proof of full abstraction uses the structure we've just described to decompose strategies into smaller ones, as we hinted at for PCF before.
First we need a few more observations:

- Any map $A \to B \oslash \mathbf{0}$ is strict: it has nowhere to go after the first move but into A.
- A map $A \& B \to \mathbf{0}$ determines either a map $A \to \mathbf{0}$, or a map $B \to \mathbf{0}$. (The converse is always true).
- A strict map $A \multimap \mathbf{0} \to B \multimap \mathbf{0}$ is either the empty strategy, or $\sigma \multimap \mathbf{0}$ for some $\sigma : B \to A$.

Another little fact

We can always construct a map

$$!(!A \multimap B) \to !A \multimap !B.$$

In our category, we can go the other way:

$$\begin{array}{rl} & !A \multimap !B \\ \to & !A \multimap !B \oslash B \\ \to & !B \oslash (!A \multimap B) \\ \to & !(!A \multimap B) \oslash (!A \multimap B) \\ \to & !(!A \multimap B) \end{array}$$

The second last step uses a constant map $!B \to !(!A \multimap B)$.

Let the decomposition begin

We'll consider a map

$$I \to !\Pi_i A_i$$

where $A_i = !\Pi_j B_{ij} \multimap \mathbf{0}$ and $B_{ij} = !\Pi_k C_{ijk} \multimap \mathbf{0}$.
Let's begin.

$$\mathcal{G}(I, !\vec{A}) \cong \mathcal{G}(I, !\vec{A} \oslash \Pi_i A_i)$$

by unpacking !.
$\oslash$ is a right adjoint and hence preserves products, so

$$\mathcal{G}(I, !\vec{A} \oslash \Pi_i A_i) \cong \mathcal{G}(I, \Pi_i(!\vec{A} \oslash A_i)) \cong \Pi_i \mathcal{G}(I, !\vec{A} \oslash A_i).$$

Decomposition

Focussing on just one element of this tuple, we continue:

$$\mathcal{G}(I, !\vec{A} \oslash A_i) = \mathcal{G}(I, !\vec{A} \oslash (!\vec{B}_i \multimap \mathbf{0})) \cong \mathcal{G}(I, !\vec{B}_i \multimap (!\vec{A} \oslash \mathbf{0}))$$

thanks to our weird isomorphism. Uncurrying, this is a map

$$!\vec{B}_i \to !\vec{A} \oslash \mathbf{0}.$$

All such maps are strict so we're now interested in

$$\mathcal{G}_s(!\vec{B}_i, !\vec{A} \oslash \mathbf{0}).$$

Decomposition

By the amazing adjunction,

$$\mathcal{G}_s(!\vec{B}_i, !\vec{A} \oslash \mathbf{0}) \cong \mathcal{G}_s(!\vec{A} \multimap !\vec{B}_i, \mathbf{0}).$$

Unpacking !, this is the same as

$$\mathcal{G}_s(!\vec{A} \multimap !\vec{B}_i \oslash \vec{B}_i, \mathbf{0}) = \mathcal{G}_s(!\vec{A} \multimap (!\vec{B}_i \oslash \Pi_j(!\vec{C}_{ij} \multimap \mathbf{0})), \mathbf{0}).$$

Again because both $\multimap$ and $\oslash$ preserve products, this is the same as

$$\mathcal{G}_s(\Pi_j(!\vec{A} \multimap (!\vec{B}_i \oslash (!\vec{C}_{ij} \multimap \mathbf{0}))), \mathbf{0}).$$

Decomposition

A map

$$\Pi_j(!\vec{A} \multimap (!\vec{B}_i \oslash (!\vec{C}_{ij} \multimap \mathbf{0}))) \longrightarrow \mathbf{0}$$

is either the empty strategy, which is definable, or picks out one of the js, so we can focus on a single map in

$$\mathcal{G}_s(!\vec{A} \multimap (!\vec{B}_i \oslash (!\vec{C}_{ij} \multimap \mathbf{0})), \mathbf{0}).$$

By the weird isomorphism, this is the same as

$$\mathcal{G}_s(!\vec{B}_i \oslash (!\vec{A} \multimap !\vec{C}_{ij} \multimap \mathbf{0}), \mathbf{0}).$$

Uncurrying we get

$$\mathcal{G}_s(!\vec{B}_i \oslash (!(\vec{A} \& \vec{C}_{ij}) \multimap \mathbf{0}), \mathbf{0}).$$

Decomposition

From

$$\mathcal{G}_s(!\vec{B}_i \oslash (!(\vec{A} \& \vec{C}_{ij}) \multimap \mathbf{0}), \mathbf{0}).$$

we can move across the amazing adjunction to

$$\mathcal{G}_s(!(\vec{A} \& \vec{C}_{ij}) \multimap \mathbf{0}, !\vec{B}_i \multimap \mathbf{0}).$$

Such a map, if not the empty strategy which we've already accounted for, is of the form $\sigma \multimap \mathbf{0}$, where σ is a strategy in

$$\mathcal{G}(!\vec{B}_i, !(\vec{A} \& \vec{C}_{ij})).$$

Decomposition

Currying gives a map

$$\mathcal{G}(I, !\vec{B}_i \multimap !(\vec{A} \& \vec{C}_{ij})).$$

which gives rise to a map

$$\mathcal{G}(I, !(!\vec{B}_i \multimap (\vec{A} \& \vec{C}_{ij})))$$

which is of the same form we started with!

Definability

Some more observations:

- When we moved from $\sigma \multimap \mathbf{0}$ to σ, we dropped two moves: the two in the $\mathbf{0}$ components.
- No step in the decomposition added any moves.
- All steps are reversible.

Decomposition as a formula

We have shown that

$$\mathcal{G}(I, !\vec{A}) \trianglelefteq \prod_i \prod_j \mathcal{G}(I, !(!\vec{B}_i \multimap (\vec{A} \& \vec{C}_{ij})))_\perp.$$

This leads to a decomposition of strategies into trees. We then show that these trees are definable—a programming problem!—and use this to prove full abstraction.

Phew.

References

[1] 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.

[2] R. Cartwright, P.-L. Curien, and M. Felleisen. Fully abstract semantics for observably sequential languages. *Information and Computation*, 111(2):297–401, 1994.

[3] Jim Laird. A categorical model of higher-order store. In *Proceedings, CTCS 2002*, 2002.