A games model of BI

Guy McCusker\textsuperscript{1}  David Pym\textsuperscript{2}

\textsuperscript{1}University of Bath  
\textsuperscript{2}HP Labs, Bristol

March 14th 2007
**BI**, the logic of bunched implications, is a substructural logic which treats multiplicative and additive versions of its connectives on an equal footing:

- **Additive** $\land$, $\rightarrow$, $;$
- **Multiplicative** $\ast$, $\ast\ast$, $,$

As a result, it gives a logical account of the notions of *sharing* and *separation* of resources.

This leads to lots of good stuff:

- a useful type system for constraining interference (O’Hearn)
- separation logic (O’Hearn, Reynolds, …)
- Hennessy-Milner style logic for resource-sensitive systems (Pym, Tofts)
αλ-calculus is the term language for the (−∗, −) fragment of BI.

Types:

\[ A ::= \gamma | A \rightarrow \ast A | A \rightarrow A, \]

Terms:

\[ M ::= x | \lambda x.M | MM | \alpha x.M | M \odot M. \]

Judgements:

\[ \Gamma \vdash M : A \]

where \( M \) is a term, \( A \) is a type, and \( \Gamma \) is a bunch.
A bunch is a tree of *identifier-type* pairs, connected by commas and semicolons:

\[ \Gamma ::= I \mid x : A \mid \Gamma, \Gamma \mid \Gamma; \Gamma. \]

Bunches are identified up to \(\equiv\), the smallest equivalence relation containing

- commutative monoid equations for \(I\) and \(;\),
- commutative monoid equations for \(I\) and \(,\),
- congruence: if \(\Delta \equiv \Delta'\) then \(\Gamma(\Delta) \equiv \Gamma(\Delta')\).

(Note for experts: we treat the *affine* version of \(\mathbf{BI}\) here.)
Typing rules: term forming

\[
\begin{align*}
\Gamma, \alpha : A & \vdash M : B \\
\Gamma & \vdash \alpha x. M : A \to B \\
\Gamma ; \alpha : A & \vdash M : B \\
\Gamma & \vdash M \otimes N : B
\end{align*}
\]
Typing rules: structural

\[ \frac{\Gamma \vdash M : A \quad \Delta \equiv \Delta}{\Gamma \equiv \Delta} \]

\[ \frac{\Gamma(\Delta) \vdash M : A \quad \Gamma(\Delta') \vdash M : A}{\Gamma(\Delta, \Delta') \vdash M : A} \quad \frac{\Gamma(\Delta) \vdash M : A}{\Gamma(\Delta; \Delta') \vdash M : A} \]

\[ \frac{\Gamma(\Delta; \Delta') \vdash M : B}{\Gamma(\Delta) \vdash M[\text{idents}(\Delta)/\text{idents}(\Delta')] : B} \quad \Delta \equiv \Delta' \]
**BI** enjoys a rich semantic theory. Models of **BI** are *cartesian doubly-closed categories*, that is, categories with two monoidal-closed structures:

- $\ast \vdash \ast$
- $\times \vdash \rightarrow$

where $\times$ is cartesian product and $\ast$ is not!

Examples of such structures:

- **Cat**
**Bi** enjoys a rich semantic theory. Models of **Bi** are *cartesian doubly-closed categories*, that is, categories with two monoidal-closed structures:

- \( \ast \vdash \ast \)
- \( \times \vdash \to \)

where \( \times \) is cartesian product and \( \ast \) is not!

Examples of such structures:

- **Cat**
- Presheaf categories and the like
**Models of BI**

**BI** enjoys a rich semantic theory.

Models of **BI** are *cartesian doubly-closed categories*, that is, categories with two monoidal-closed structures:

- $\otimes \dashv \multimap$
- $\times \dashv \rightarrow$

where $\times$ is cartesian product and $\otimes$ is not!

Examples of such structures:

- **Cat**
- Presheaf categories and the like
- **Err, ...**
BI enjoys a rich semantic theory. Models of BI are cartesian doubly-closed categories, that is, categories with two monoidal-closed structures:

- $\bullet \; * \; \rightarrow *$
- $\bullet \; \times \; \rightarrow$

where $\times$ is cartesian product and $*$ is not!

Examples of such structures:

- Cat
- Presheaf categories and the like
- Err, ...
- That is all.
A new model of **BI**!

In this talk we present a new kind of model of **BI**: a game semantics.

- Good news: it works.
In this talk we present a new kind of model of **BI**: a game semantics.

- Good news: it works.
- Bad news: it took me nine years.
In this talk we present a new kind of model of \textbf{BI}: a game semantics.

- Good news: it works.
- Bad news: it took me nine years.
- Good news: it’s not \textbf{Cat} and it’s not a functor category.
A new model of BI!

In this talk we present a new kind of model of BI: a game semantics.

- Good news: it works.
- Bad news: it took me nine years.
- Good news: it’s not Cat and it’s not a functor category.
- Bad news: it’s not a cartesian DCC either.
In this talk we present a new kind of model of **BI**: a game semantics.

- Good news: it works.
- Bad news: it took me nine years.
- Good news: it’s not **Cat** and it’s not a functor category.
- Bad news: it’s not a cartesian DCC either.
- Good news: it’s *fully complete*. 
Game semantics models computation or proof as *interaction* between two characters, O and P.

P represents the system, the program, the proof.

O represents the environment, the user, the anti-proof.

They take turns to make moves which constitute a discussion of the term/system/proof in question.

The type or formula dictates what moves are allowed.
Game semantics = abstract Böhm trees

Consider a Böhm-tree such as

$$\lambda x_1 x_2. x_2 \ M_1 \ (\lambda y_1 y_2. x_1 \ y_1).$$

The game-semantics of this is something like:

- **O**: What is the head variable of this term?
Consider a Böhm-tree such as

\[ \text{\(\lambda x_1 x_2 . x_2 \ M_1 (\lambda y_1 y_2 . x_1 y_1)\).} \]

The game-semantics of this is something like:

- O: What is the head variable of this term?
- P: It’s \(x_2\), abstracted at the top level.
Consider a Böhm-tree such as

\[ \lambda x_1 x_2. x_2 M_1 (\lambda y_1 y_2. x_1 y_1). \]

The game-semantics of this is something like:

- O: What is the head variable of this term?
- P: It’s \( x_2 \), abstracted at the top level.
- O: What is the head-variable of the second argument to this \( x_2 \)?
Consider a Böhm-tree such as

$$\lambda x_1 x_2 . x_2 \ M_1 \ (\lambda y_1 y_2 . x_1 \ y_1).$$

The game-semantics of this is something like:

- O: What is the head variable of this term?
- P: It's $x_2$, abstracted at the top level.
- O: What is the head-variable of the second argument to this $x_2$?
- P: It's $x_1$, abstracted at the top level.
Consider a Böhm-tree such as

$$\lambda x_1 x_2 . x_2 \ M_1 (\lambda y_1 y_2 . x_1 \ y_1) .$$

The game-semantics of this is something like:

- O: What is the head variable of this term?
- P: It’s $x_2$, abstracted at the top level.
- O: What is the head-variable of the second argument to this $x_2$?
- P: It’s $x_1$, abstracted at the top level.
- O: What is the head variable of the first argument to this $x_1$?
Consider a Böhm-tree such as

$$\lambda x_1 x_2. x_2 \; M_1 \; (\lambda y_1 y_2. x_1 \; y_1).$$

The game-semantics of this is something like:

- **O:** What is the head variable of this term?
  - **P:** It’s $x_2$, abstracted at the top level.
- **O:** What is the head-variable of the second argument to this $x_2$?
  - **P:** It’s $x_1$, abstracted at the top level.
- **O:** What is the head variable of the first argument to this $x_1$?
  - **P:** It’s $y_1$, abstracted at the second level.
More abstractly...

\[ \lambda x_1 x_2. x_2 \ M_1 (\lambda y_1 y_2. x_1 \ y_1). \]

\[(\gamma_{1,1} \rightarrow \gamma_1) \rightarrow (\gamma_{2,1} \rightarrow (\gamma_{2,2,1} \rightarrow \gamma_{2,2,2} \rightarrow \gamma_{2,2}) \rightarrow \gamma_2) \rightarrow \gamma \]
Types are modelled as *games*: sets of moves, with rules saying which moves can be played when. Each move comes with a justification pointer.

A term is modelled as a *strategy*: a preordained collection of responses by P to the possible moves O might make.
When modelling the $\lambda$-calculus, it turns out that:

- the game interpreting a type has the atomic subtypes as moves
- the justification pointer of a move indicates its parent in the type-tree
- if O always moves to a direct descendant of P’s last move, plays can be seen as Böhm tree branches
- in that case, the strategy interpreting a term simply describes the Böhm tree of that term.
Generalizing

The condition that $O$ always moves to a direct descendent of the last $P$-move is too restrictive:

- it’s asymmetric, so we can’t let strategies play against one another; this means we can’t properly interpret computation
- it forces plays to look like Böhm-tree branches, trapping us in the $\lambda$-calculus
We’d like to model substitution via composition of strategies, and we’d like that to involve the *interaction* of two strategies, cf. “parallel composition with hiding” in processes.

For instance: $f(\lambda a.a)(\lambda b.b)[(\lambda xy.yx)/f]$.

First let’s look at $f(\lambda a.a)(\lambda b.b)$:
Composition as interaction

Now let’s look at $\lambda xy.yx$:

$$(\gamma_{1,1,1} \rightarrow \gamma_{1,1}) \rightarrow (\gamma_{1,2,1} \rightarrow \gamma_{1,2}) \rightarrow \gamma_{1,1,1}$$
Now play them off against one another:

$$f(\lambda a.a)(\lambda b.b)[\lambda xy.yx/f]$$

$$(\gamma_{1,1,1} \rightarrow \gamma_{1,1}) \rightarrow (\gamma_{1,2,1} \rightarrow \gamma_{1,2}) \rightarrow \gamma_1 \rightarrow \gamma$$

At $\gamma_{1,1}$, we need to get a response from $f(\lambda a.a)(\lambda b.b)$. But it doesn’t yet have a response to this kind of play, since here $\gamma_{1,1}$ is not interrogating the most recent move.
The solution is to *collapse* the play so that it looks like O always interrogates the most recent move: the view $\text{view}(s)$ of a play is defined by

\[
\text{view}(\varepsilon) = \varepsilon
\]
\[
\text{view}(s \cdot a) = a, \text{ if } a \text{ is an initial move}
\]
\[
\text{view}(s \cdot a \cdot t \cdot b) = \text{view}(s) \cdot a \cdot b.
\]

We set things up so that a strategy depends only on the view that it sees: these are called *innocent* strategies.

Clearly, an innocent strategy is fully determined by the set of responses it gives at view-like positions, i.e. positions where O always interrogates the last P-move.
Innocent strategies are abstract Böhm trees

An innocent strategy is determined by a set of “view like plays”.

A view-like play corresponds to a branch of a Böhm-tree.

Hence strategies correspond to potentially partial, potentially infinite Böhm-trees.
The “abstract Böhm tree” reading of game semantics gives us the following intuitive understanding of interactions:

- An O-move chooses a subterm to interrogate. The initial move interrogates the whole term. Later moves interrogate particular arguments to particular variables.
- A P-move chooses a head-variable for a subterm. The justification pointer indicates where that variable was abstracted.
- If the pointer from \( m \) points past a move \( n \), that means the head-variable indicated by \( m \) appears free in the subterm indicated by \( n \).
Categories of games are typically constructed as follows.

- Objects: games.
- Arrows $A \rightarrow B$: strategies on a game $A \vdash B$.
- Composition: interaction of strategies.
- Identities: copycat strategies.
The copycat strategy copies moves from its domain to its codomain and vice versa.

\[ ((\gamma'_1 \to \gamma'_2) \to \gamma'_3) \vdash ((\gamma_1 \to \gamma_2) \to \gamma_3) \]

\[ \gamma'_1 \]
\[ \gamma'_2 \]
\[ \gamma'_3 \]
\[ \gamma_1 \]
\[ \gamma_2 \]
\[ \gamma_3 \]

Intuition check: this corresponds to \( \lambda f. g(\lambda x. fx) =_\eta \lambda f. gf =_\eta g \).
Full completeness for the $\lambda$-calculus

Theorem

Games and innocent strategies form a CCC, and hence a model of the $\lambda$-calculus. If $A$ is a type, $\Gamma$ a context, and $\sigma : \llbracket \Gamma \rrbracket \to \llbracket A \rrbracket$ a finite, total innocent strategy, then there exists a Böhm-tree $\Gamma \vdash M : A$ such that $\llbracket M \rrbracket = \sigma$.

This is the key to the full abstraction theorems for Hyland-Ong/Nickau’s game semantics of PCF. We will now show how to refine it for $\alpha\lambda$-calculus.
The $\alpha \lambda$-calculus refines the $\lambda$-calculus in two ways:

- there are two kinds of function type, $\to$ and $\multimap$
- contraction is constrained to work across semicolon only. This restricts what terms are typeable.

We therefore need to refine our game model in two ways:

- distinguish between the additive and multiplicative arrows, and
- constrain the behaviour of our strategies appropriately.
Adding information to the types

We add a simple notion of separation relation to our games: \( \# \) is a relation between moves of a game which will reflect the presence of multiplicative connectives in the types.

In \( \gamma_1 \to (\gamma_2 \rightarrow \gamma_3) \), \( \gamma_1 \) and \( \gamma_2 \) are arguments which cannot share resources, so we have \( \gamma_1 \# \gamma_2 \).

Furthermore, the function itself may not share resources with \( \gamma_2 \), so we also have \( \gamma_2 \# \gamma_3 \).

The notion of “simple type plus separation relation” is more than enough to express the types of \( \alpha \lambda \)-calculus; in fact it is more general. (Atkey’s \( \lambda_{sep} \)?)
Some syntactic facts

The only important information in the bunch structure is in fact the separation relation, for the following reasons:

**Lemma (Additive application)**

If $\Gamma \vdash M : A \rightarrow B$ and $\Gamma \vdash N : A$ then $\Gamma \vdash M \otimes N : B$.

**Lemma (Multiplicative application)**

A term $\Gamma \vdash MN : B$ is typeable if and only if there are typeable terms $\Gamma \vdash M : A \multimap B$ and $\Gamma \vdash N : A$, and all the free identifiers of $M$ are separated from all the free identifiers of $N$ in $\Gamma$. 
Consider a Böhm-tree \( \Gamma \vdash fM_1 \ldots M_n : A \) in the \( \lambda \)-calculus.

Augment the context \( \Gamma \) and the type \( A \) with separation relations.

Under what circumstances is \( fM_1 \ldots M_n \) typeable in \( \alpha \lambda \)-calculus?
By our lemmas, the following constraints suffice:

- if $M_i$ and $M_j$ are arguments in separated parts of the type, then $M_i$’s free variables must be separated from $M_j$’s.
- if $M_i$ is an argument to a part of the type separated from the head-variable $f$, then its free variables must be separated from $f$
- we must be able to type each $\Gamma \vdash M_i$. 
Suppose $M_i$ has type $\gamma_1 \multimap \gamma_2 \rightarrow \gamma_3 \multimap \gamma$. Then $M_i$ has the form

$$\Gamma \vdash \lambda x. \alpha y. \lambda z. M'_i : \gamma_1 \multimap \gamma_2 \rightarrow \gamma_3 \multimap \gamma.$$ 

The type immediately tells us that $z$ is separated from the other variables, and $x$ is separated from the function. But there's more. We need to type

$$((\Gamma, x); y), z \vdash M'_i : \gamma.$$ 

Therefore, when considering this subterm, its $\lambda$-abstracted variables are separated from $\Gamma$. 

*McCusker and Pym (U. Bath, HP Bristol)*

March 14th 2007 29 / 43
Let’s try to turn these ideas into game-semantic ones. First, recall that a “free variable in a term” corresponds to a P-move $n$ whose justifier is before a previous O-move $m$.

**Definition**

Given a play $s$ containing an O-move $m$ and later P-move $n$, we write $n \text{ ext}_s m$ iff $m \in \text{view}(s_{<n})$ and $n$’s justifier is in $s_{<m}$.
Non-sharing arguments

Now we can begin to transport our intuition to the semantic setting.

if $M_i$ and $M_j$ are arguments in separated parts of the type, then $M_i$’s free variables must be separated from $M_j$’s

becomes

if $m_1 \neq m_2$, and $n_i \text{ ext } m_i$ for $i = 1, 2$, then $n_1 \neq n_2$
if $M_i$ is an argument to a part of the type separated from the head-variable $f$, then its free variables must be separated from $f$ becomes

if $m$ is justified by $m'$ with $m \not\equiv m'$, and n $\text{ext}$ $m$, then $n \not\equiv m'$. 
We’ve forgotten about what happens when we dig into subterms:

\textit{when considering a subterm, its }\lambda\text{-abstracted variables are separated from }\Gamma.\textit{

Capture this with another definition:

\textbf{Definition}

Given a play \(s\) containing moves \(m\) and \(n\), we write \(m \ast_s n\) iff any of the following conditions holds:

- \(m \#_s n\);
- \(m\) is justified by \(m'\), \(n\ ext m'\) and \(m \#_A m'\); or
- \(n\) is justified by \(n'\), \(m\ ext n'\) and \(n \#_A n'\).
In a picture: $m \star n$ if

\[\begin{array}{c}
\# \\
\vdots \\
m' \\
m
\end{array}\]
A play $s$ is separation safe if:

- for any O-moves $m_1, m_2 \in s$ with $m_1 \not\#_s m_2$, if $n_1 \text{ext } m_1$ and $n_2 \text{ext } m_2$ then $n_1 \ast_s n_2$.
- for any P-move $n$ such that $n \text{ext } m$ in $s$, if $m$ is justified by $m'$ and $m \not\#_A m'$ then $n \ast_s m'$.

A strategy is separation safe if all its plays are.
The category $G_{\text{sep}}$ has:

- Objects: games with separation relations
- Maps: separation safe strategies
- Composition and identities are as normal.
Proving that separation safety is preserved by composition is hard work. If $n \texttt{ext} m$ in a composite play, there must be $n_1, \ldots, n_k$ in the hidden part such that

$$n \texttt{ext} n_1 \texttt{ext} \cdots \texttt{ext} n_k \texttt{ext} m.$$ 

\[ \textbf{Lemma} \]

Let $s$ be a play containing moves $m_1, m_2$ and $n_{i,1}, \ldots, n_{i,k_i}$ for $i = 1, 2$ with

$$n_{i,1} \texttt{ext} n_{i,2} \texttt{ext} \cdots \texttt{ext} n_{i,k_i} \texttt{ext} m_i$$

and $m_1 \ast m_2$, then $n_1 \ast n_2$. 
Case 1

$m_1 \ast m_2$ because $m_1 \not\approx m_2$:

\[
\begin{array}{c}
\vdots \\
m_1 \\
\text{ext} \\
\vdots \\
\text{ext} \\
\vdots \\
n_1,1
\end{array}
\quad
\begin{array}{c}
\vdots \\
\not\approx \\
j \\
\vdots \\
\text{ext} \\
\vdots \\
n_2,1
\end{array}
\quad
\begin{array}{c}
\vdots \\
m_2 \\
\text{ext} \\
\vdots \\
\text{ext} \\
\vdots \\
n_2,2
\end{array}
\]

\[n_{1,k_1} \quad n_{2,k_2}\]
Case 1

$m_1 * m_2$ because $m_1 \neq m_2$:

\[
\begin{array}{ccc}
\vdots & \vdots & \\
& j & \\
\circlearrowright m_1 \# m_2 \circlearrowleft \\
\text{ext} & \text{ext} & \\
\circlearrowright n_{1,k_1} \# n_{2,k_2} \circlearrowleft \\
\text{ext} & \text{ext} & \\
\vdots & \vdots & \\
\text{ext} & \text{ext} & \\
n_{1,1} & n_{2,1} & \\
\end{array}
\]
Case 2

$m_1 \ast m_2$ because $m_1$ justified by $m'_1$, $m'_1 \not\equiv m_1$ and $m_2 \text{ext } m'_1$:

\[
\begin{array}{c}
\vdots & \vdots \\
\# & m'_1 \\
m_1 & m_2 \\
\text{ext} & \text{ext} \\
n_{1,k_1} & n_{2,k_2} \\
\text{ext} & \text{ext} \\
\vdots & \vdots \\
\text{ext} & \text{ext} \\
n_{1,1} & n_{2,1}
\end{array}
\]
Case 2

\[ m_1 \ast m_2 \text{ because } m_1 \text{ justified by } m'_1, \ m'_1 \not\equiv m_1 \text{ and } m_2 \text{ ext } m'_1: \]

\[ \begin{array}{c}
\vdots \quad \vdots \\
\# \quad m'_1 \\
m_1 \quad m_2 \\
\text{ext} \quad \text{ext} \\
n_{1,k_1} \quad n_{2,k_2} \\
\text{ext} \quad \text{ext} \\
\vdots \quad \vdots \\
\text{ext} \quad \text{ext} \\
n_{1,1} \quad n_{2,1} \\
\end{array} \Rightarrow \]
Case 2

$m_1 \ast m_2$ because $m_1$ justified by $m'_1$, $m'_1 \not\approx m_1$ and $m_2 \text{ ext } m'_1$:

\[
\begin{array}{ccc}
\vdots & \vdots & \\
\# & m'_1 & \\
\downarrow & \downarrow & \\
\text{ext} & \text{ext} & \Rightarrow \\
n_{1,k_1} & n_{2,k_2} & \\
\text{ext} & \text{ext} & \\
\vdots & \vdots & \\
\text{ext} & \text{ext} & \\
n_{1,1} & n_{2,1} & \\
\end{array}
\]
Wrapping up

In $\mathcal{G}_{\text{sep}}$, the product from the original games model splits in two: $*$ places a $\#$ relation between the two parts, while $\times$ does not. Thus we have two monoidal structures.

Likewise, $\Rightarrow$ splits in two: $\rightarrow$ places a $\#$ between the two parts while $\Rightarrow$ does not.

We have

\[
\mathcal{G}_{\text{sep}}(A \ast B, C) \cong \mathcal{G}_{\text{sep}}(A, B \rightarrow C) \\
\mathcal{G}_{\text{sep}}(A \times B, C) \cong \mathcal{G}_{\text{sep}}(A, B \rightarrow C)
\]

but, sadly, not all $A \rightarrow B$ objects exist.
$G_{\text{sep}}$ is cartesian closed, has another monoidal structure $\ast$, and $\ast$ has exponentials $A \rightarrow^\ast B$ for a large class of objects $B$; so it’s almost a CDCC.

All the $\rightarrow^\ast$-types needed to model $\alpha\lambda$-calculus are available.
Theorem

If $A$ is a type, $\Gamma$ a bunch, and $\sigma$ a separation-safe finite, total innocent strategy on $\Gamma \vdash A$, then there is a term $\Gamma \vdash M : A$ of $\alpha \lambda$-calculus such that $\sigma = \llbracket M \rrbracket$.

Proof  We can find a term $M$ of ordinary $\lambda$-calculus such that $\llbracket M \rrbracket = \sigma$. Separation safety of $\sigma$ guarantees that the typing constraints of $\alpha \lambda$-calculus are satisfied.
Where next?

Lots of things to do:

- check the correspondence with $\lambda_{sep}$.
- move beyond innocence: model imperative programming, SCI+.
- move beyond views: can we make sense of this in the most general setting, where pointers can be modelled?
- get rid of weakening! Melliès’s work might help.
- ...