

A games model of BI

Guy McCusker¹ David Pym²

¹University of Bath

²HP Labs, Bristol

March 14th 2007

The logic of bunched implications

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

Additive $\wedge, \rightarrow, ;$

Multiplicative $*, \multimap, ,$

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)

$\alpha\lambda$ -calculus is the term language for the (\multimap, \rightarrow) fragment of **BI**.

Types:

$$A ::= \gamma \mid A \multimap A \mid A \rightarrow A,$$

Terms:

$$M ::= x \mid \lambda x.M \mid MM \mid \alpha x.M \mid M @ M.$$

Judgements:

$$\Gamma \vdash M : A$$

where M is a term, A is a type, and Γ 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 **BI** here.)

Typing rules: term forming

$$\frac{}{x : A \vdash x : A}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x. M : A \multimap B}$$

$$\frac{\Gamma \vdash M : A \multimap B \quad \Delta \vdash N : A}{\Gamma, \Delta \vdash MN : B}$$

$$\frac{\Gamma; x : A \vdash M : B}{\Gamma \vdash \alpha x. M : A \rightarrow B}$$

$$\frac{\Gamma \vdash M : A \rightarrow B \quad \Delta \vdash N : A}{\Gamma; \Delta \vdash M @ N : B}$$

Typing rules: structural

$$\frac{\Gamma \vdash M : A}{\Delta \vdash M : A} \Gamma \equiv \Delta$$

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

$$\frac{\Gamma(\Delta; \Delta') \vdash M : B}{\Gamma(\Delta) \vdash M[\text{idents}(\Delta)/\text{idents}(\Delta')] : B} \Delta \cong \Delta'$$

BI enjoys a rich semantic theory.

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

▶ $* \dashv \multimap$

▶ $\times \dashv \rightarrow$

where \times is cartesian product and $*$ 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:

▶ $* \dashv \multimap$

▶ $\times \dashv \rightarrow$

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

Examples of such structures:

▶ **Cat**

▶ Presheaf categories and the like

BI enjoys a rich semantic theory.

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

▶ $* \dashv \multimap$

▶ $\times \dashv \rightarrow$

where \times is cartesian product and $*$ 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:

▶ $* \dashv \multimap$

▶ $\times \dashv \rightarrow$

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.

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.

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.

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.

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.
- ▶ Good news: it's *fully complete*.

Game semantics: a quick introduction

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.

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

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

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

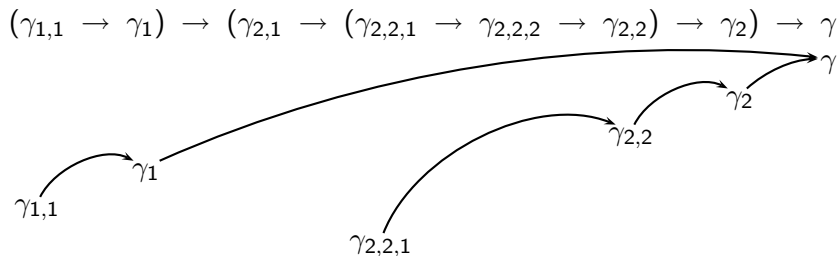
$$\lambda x_1 x_2. x_2 M_1 (\lambda y_1 y_2. x_1 y_1).$$

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...

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



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 λ -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 descendent 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.

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 λ -calculus

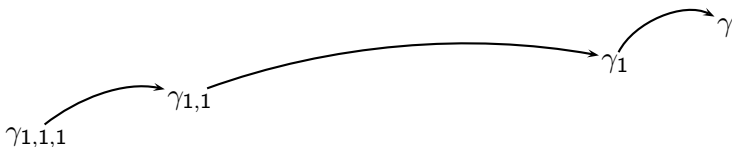
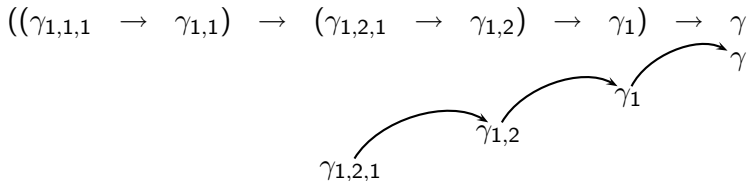
▶ [Jump to views](#)

Composition as interaction

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.

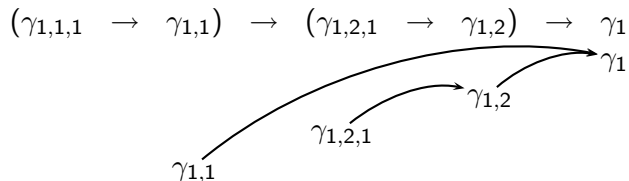
For instance: $f(\lambda a.a)(\lambda b.b)[(\lambda xy.yx)/f]$.

First let's look at $f(\lambda a.a)(\lambda b.b)$:



Composition as interaction

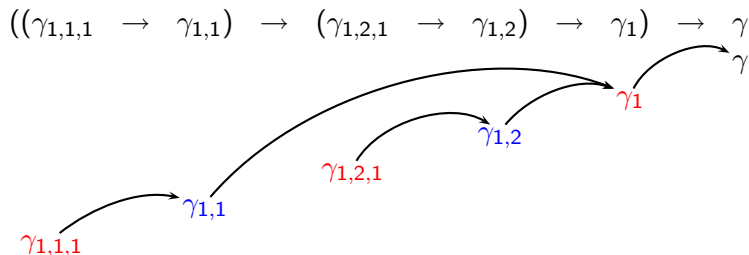
Now let's look at $\lambda xy.yx$:



Composition as interaction

Now play them off against one another:

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



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

$$\begin{aligned}\text{view}(\varepsilon) &= \varepsilon \\ \text{view}(s \cdot a) &= a, \text{ if } a \text{ is an initial move} \\ \text{view}(s \cdot \overbrace{a \cdot t} \cdot b) &= \text{view}(s) \cdot \overbrace{a} \cdot b.\end{aligned}$$

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.

Abstract Böhm trees: intuition

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

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.

Copycat strategy

The copycat strategy copies moves from its domain to its codomain and vice versa.

$$\begin{array}{ccccccc} ((\gamma'_1 \rightarrow \gamma'_2) \rightarrow \gamma'_3) & \vdash & ((\gamma_1 \rightarrow \gamma_2) \rightarrow \gamma_3) \\ & & \gamma_3 \\ & & \gamma'_3 \\ & & \gamma'_2 \\ & & \gamma_2 \\ & & \gamma_1 \\ & & \gamma'_1 \end{array}$$

Intuition check: this corresponds to $\lambda f.g(\lambda x.fx) =_{\eta} \lambda f.gf =_{\eta} g$.

Theorem

Games and innocent strategies form a CCC, and hence a model of the λ -calculus. If A is a type, Γ a context, and $\sigma : \llbracket \Gamma \rrbracket \rightarrow \llbracket A \rrbracket$ 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 λ -calculus in two ways:

- ▶ there are two kinds of function type, \rightarrow 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 $\gamma_1 \rightarrow (\gamma_2 \multimap \gamma_3)$, γ_1 and γ_2 are arguments which cannot share resources, so we have $\gamma_1 \# \gamma_2$.

Furthermore, the function itself may not share resources with γ_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 $\lambda_{\text{sep}}?$)

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 $\Gamma \vdash M @ N : B$.

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 Γ .

Constraining terms

Consider a Böhm-tree $\Gamma \vdash fM_1 \dots M_n : A$ in the λ -calculus.

Augment the context Γ and the type A with separation relations.

Under what circumstances is $fM_1 \dots M_n$ 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 $\Gamma \vdash M_i$.

Typing the subterms

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 λ -abstracted variables are separated from Γ .*

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 $n \mathbf{ext}_s m$ 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 $m_1 \# m_2$, and $n_i \mathbf{ext} m_i$ for $i = 1, 2$, then $n_1 \# n_2$

Functions which do not share with their arguments

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 $m \# m'$, and $n \mathbf{ext} m$, then $n \# m'$.

is not enough

We've forgotten about what happens when we dig into subterms:

when considering a subterm, its λ -abstracted variables are separated from Γ .

Capture this with another definition:

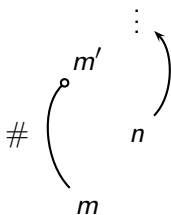
Definition

Given a play s containing moves m and n , we write $m *_s n$ iff any of the following conditions holds:

- ▶ $m \#_s n$;
- ▶ m is justified by m' , $n \text{ ext } m'$ and $m \#_A m'$; or
- ▶ n is justified by n' , $m \text{ ext } n'$ and $n \#_A n'$.

* illustrated

In a picture: $m * n$ if



Definition

A play s is *separation safe* if:

- ▶ for any O-moves $m_1, m_2 \in s$ with $m_1 \#_s m_2$, if $n_1 \mathbf{ext} m_1$ and $n_2 \mathbf{ext} m_2$ then $n_1 *_s n_2$.
- ▶ for any P-move n such that $n \mathbf{ext} m$ in s , if m is justified by m' and $m \#_A m'$ then $n *_s m'$.

A strategy is separation safe if all its plays are.

The category \mathcal{G}_{sep} has:

- ▶ Objects: games with separation relations
- ▶ Maps: separation safe strategies
- ▶ Composition and identities are as normal.

▶ Skip to wrapping-up

Composition of separation safe strategies

Proving that separation safety is preserved by composition is hard work. If $n \text{ ext } m$ in a composite play, there must be n_1, \dots, n_k in the hidden part such that

$$n \text{ ext } n_1 \text{ ext } \dots \text{ ext } n_k \text{ ext } m.$$

Lemma

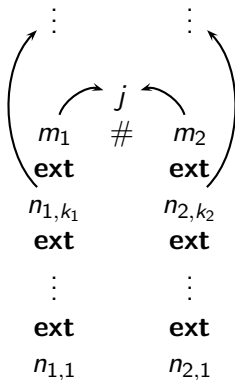
Let s be a play containing moves m_1, m_2 and $n_{i,1}, \dots, n_{i,k_i}$ for $i = 1, 2$ with

$$n_{i,1} \text{ ext } n_{i,2} \text{ ext } \dots \text{ ext } n_{i,k_i} \text{ ext } m_i$$

and $m_1 * m_2$, then $n_1 * n_2$.

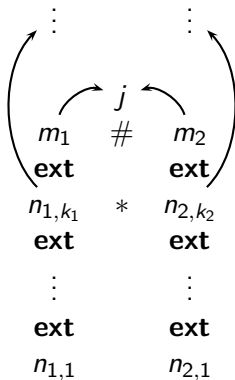
Case 1

$m_1 * m_2$ because $m_1 \# m_2$:



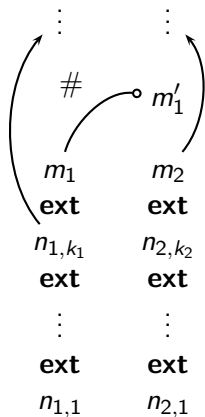
Case 1

$m_1 * m_2$ because $m_1 \# m_2$:



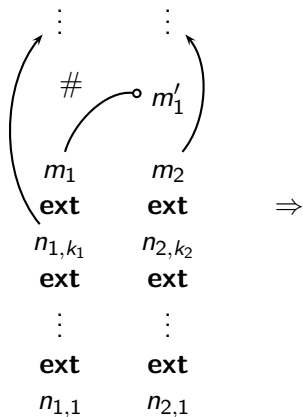
Case 2

$m_1 * m_2$ because m_1 justified by m'_1 , $m'_1 \# m_1$ and $m_2 \text{ ext } m'_1$:



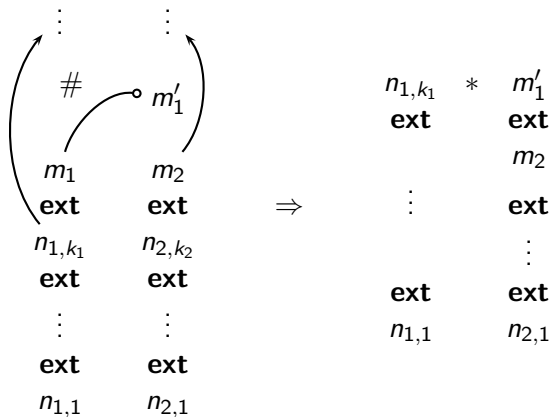
Case 2

$m_1 * m_2$ because m_1 justified by m'_1 , $m'_1 \# m_1$ and $m_2 \text{ ext } m'_1$:



Case 2

$m_1 * m_2$ because m_1 justified by m'_1 , $m'_1 \# m_1$ and $m_2 \text{ ext } m'_1$:



Wrapping up

In \mathcal{G}_{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: \multimap places a $\#$ between the two parts while \rightarrow does not.

We have

$$\begin{aligned}\mathcal{G}_{\text{sep}}(A * B, C) &\cong \mathcal{G}_{\text{sep}}(A, B \multimap C) \\ \mathcal{G}_{\text{sep}}(A \times B, C) &\cong \mathcal{G}_{\text{sep}}(A, B \rightarrow C)\end{aligned}$$

but, sadly, not all $A \multimap B$ objects exist.

\mathcal{G}_{sep} is cartesian closed, has another monoidal structure $*$, and $*$ has exponentials $A \multimap B$ for a large class of objects B ; so it's almost a CDCC.

All the \multimap -types needed to model $\alpha\lambda$ -calculus are available.

Theorem

If A is a type, Γ a bunch, and σ 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 λ -calculus such that $\llbracket M \rrbracket = \sigma$. Separation safety of σ guarantees that the typing constraints of $\alpha\lambda$ -calculus are satisfied. □

Where next?

Lots of things to do:

- ▶ check the correspondence with λ_{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.
- ▶ ...