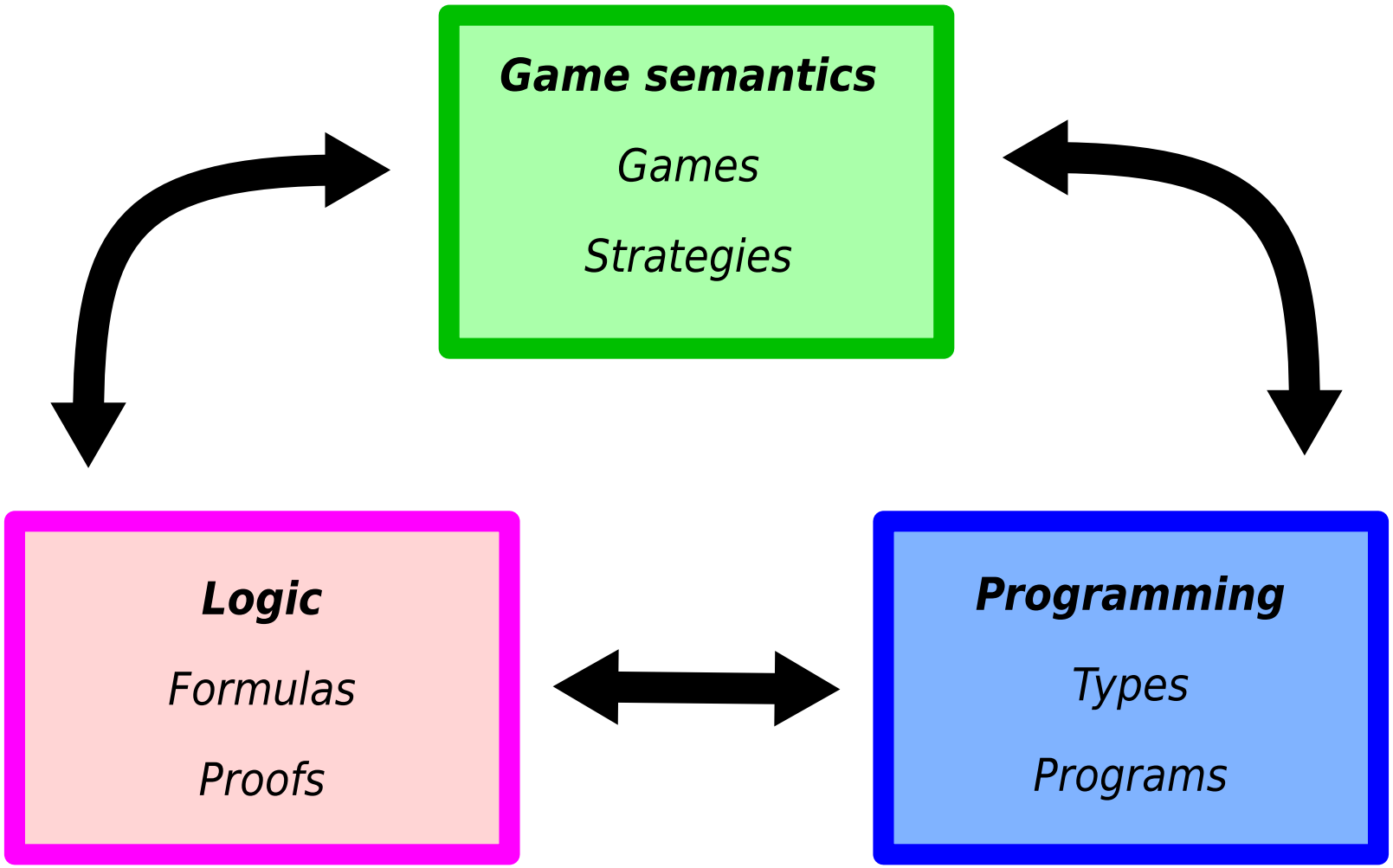
Modalités de ressource et contrôle en logique tensorielle

soutenance de thèse de
Nicolas Tabareau

PPS (Preuves Programmes Systèmes)
PARIS, FRANCE

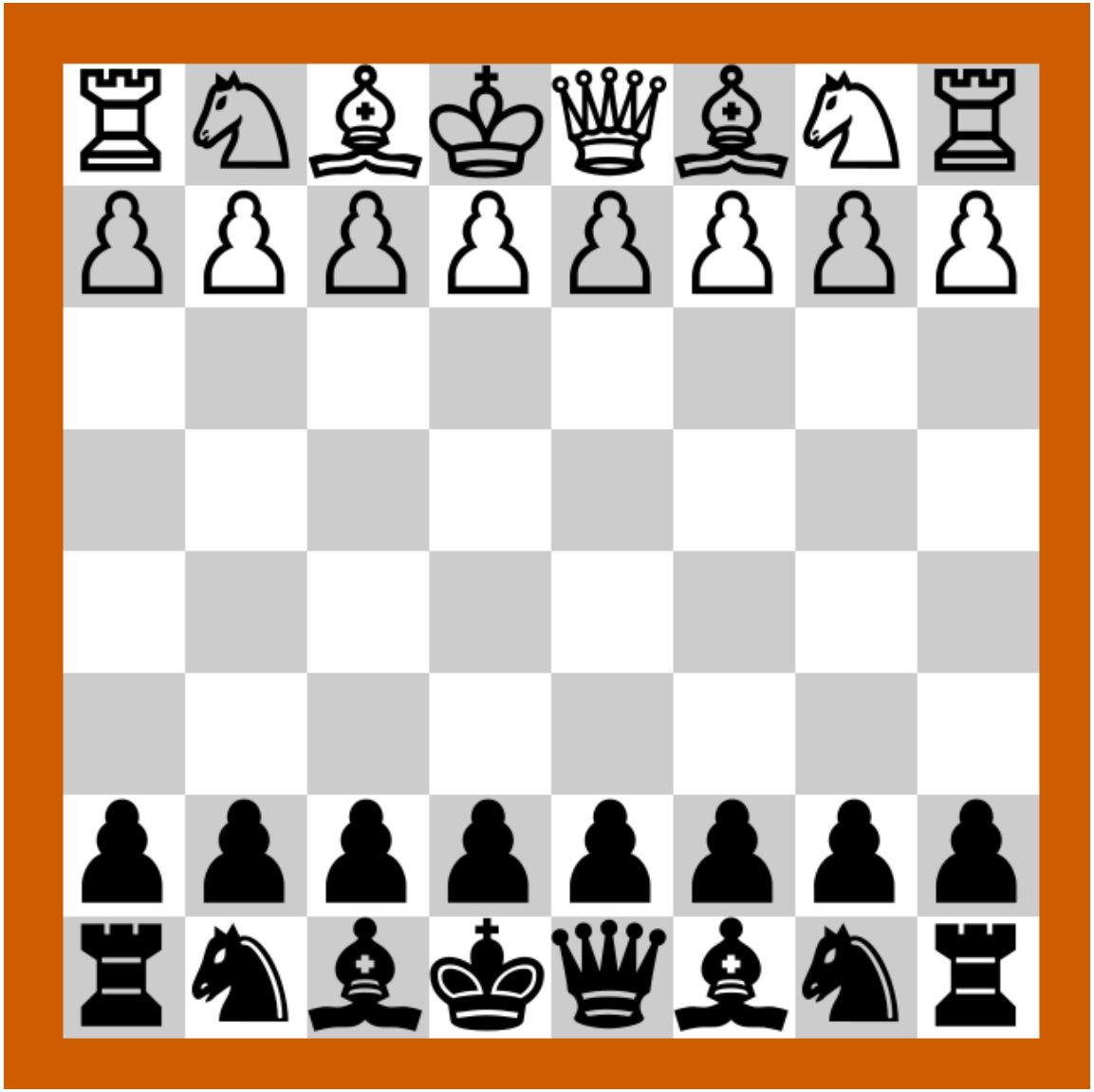
3 décembre 2008

Denotational semantics triptych



Running example : chess game

Chess game : negation $A \mapsto \neg A$



Chess game : tensor product $A \otimes B$

$\otimes$

Chess game : linear implication $A \multimap B$

$$A \multimap B \stackrel{\text{def}}{=} \neg(A \otimes \neg B)$$

Chess game : copycat strategies

Example : the copycat strategy

Chess $\multimap$ Chess

Chess game : copycat strategies

Example : the copycat strategy

Chess $\multimap$ Chess

Chess game : copycat strategies

Example : the copycat strategy

Chess $\multimap$ Chess

Chess game : copycat strategies

Example : the copycat strategy

Chess $\multimap$ Chess

Chess game : copycat strategies

Example : the copycat strategy

Chess $\multimap$ Chess

Chess game : copycat strategies

Example : the copycat strategy

Chess $\multimap$ Chess

Chess game : copycat strategies

Example : the copycat strategy

Chess $\multimap$ Chess

Chess game : copycat strategies

Example : the copycat strategy

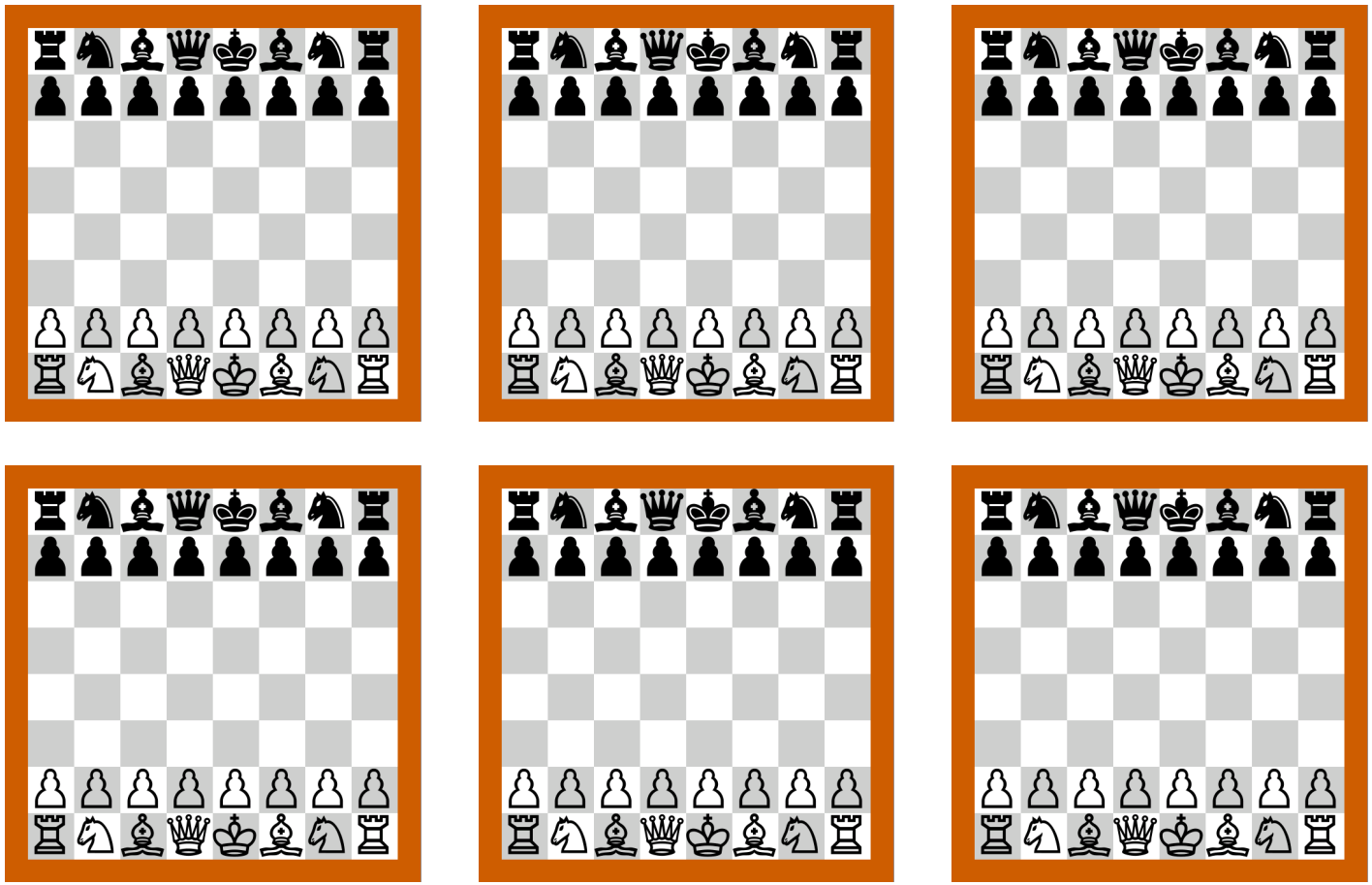
Chess $\multimap$ Chess

Chess game : copycat strategies

Example : the copycat strategy

Chess $\multimap$ Chess

Chess game : exponential $A \mapsto !A$



A logic for games

How do all those game constructions combine ?

A logic for games

We will develop a logic with

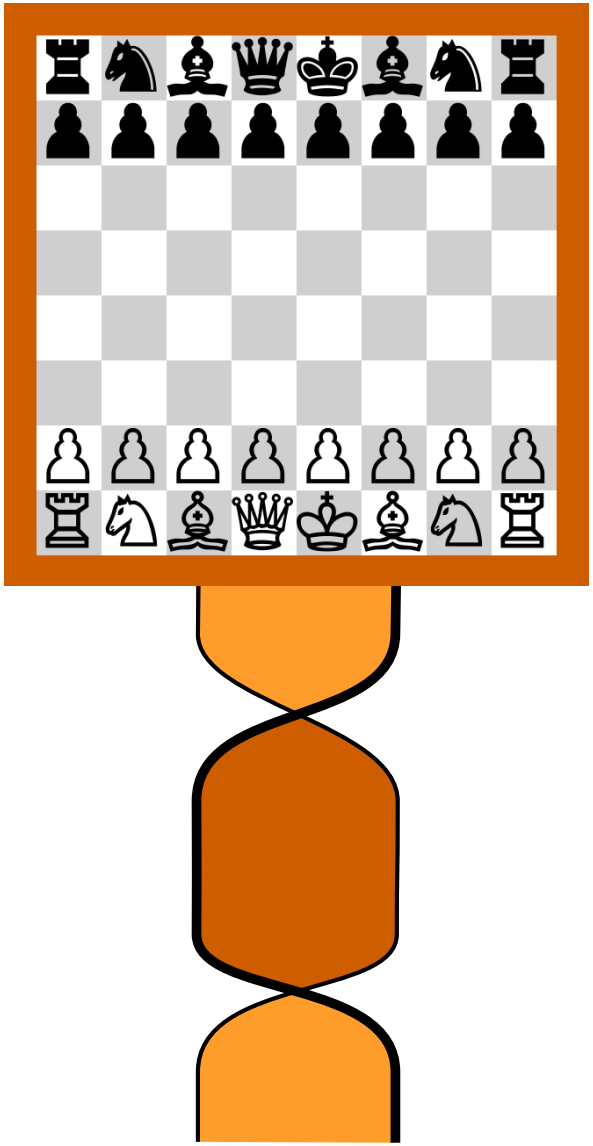
$$\neg A \quad | \quad A \otimes B \quad | \quad !A$$

The isomorphism of linear logic

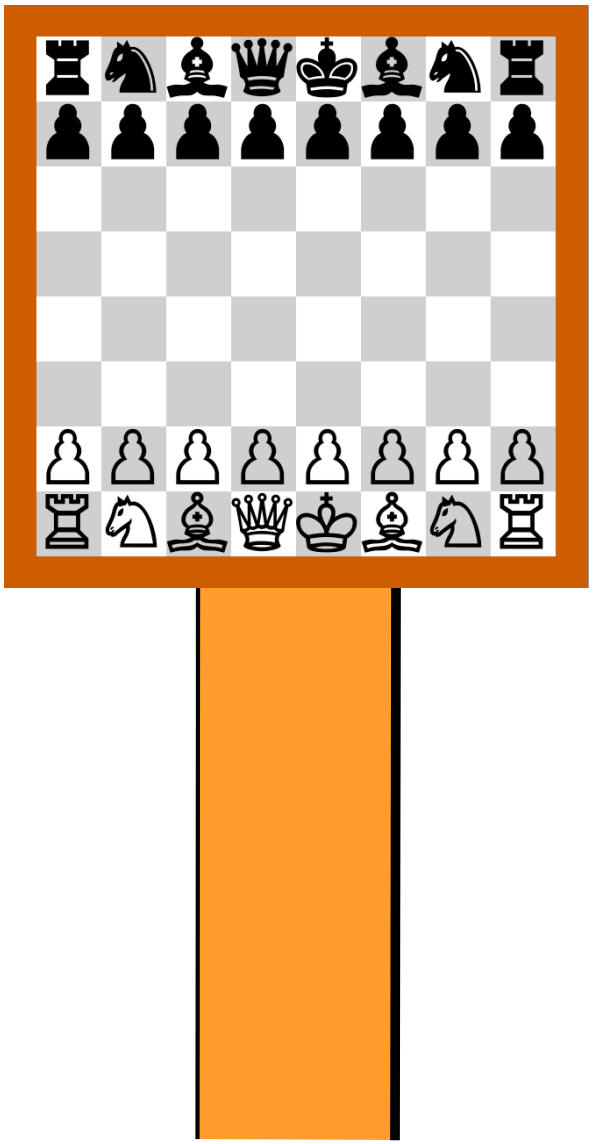
$$A \cong \neg\neg A$$

requires to identify strategies in game models. . . we thus remove it !

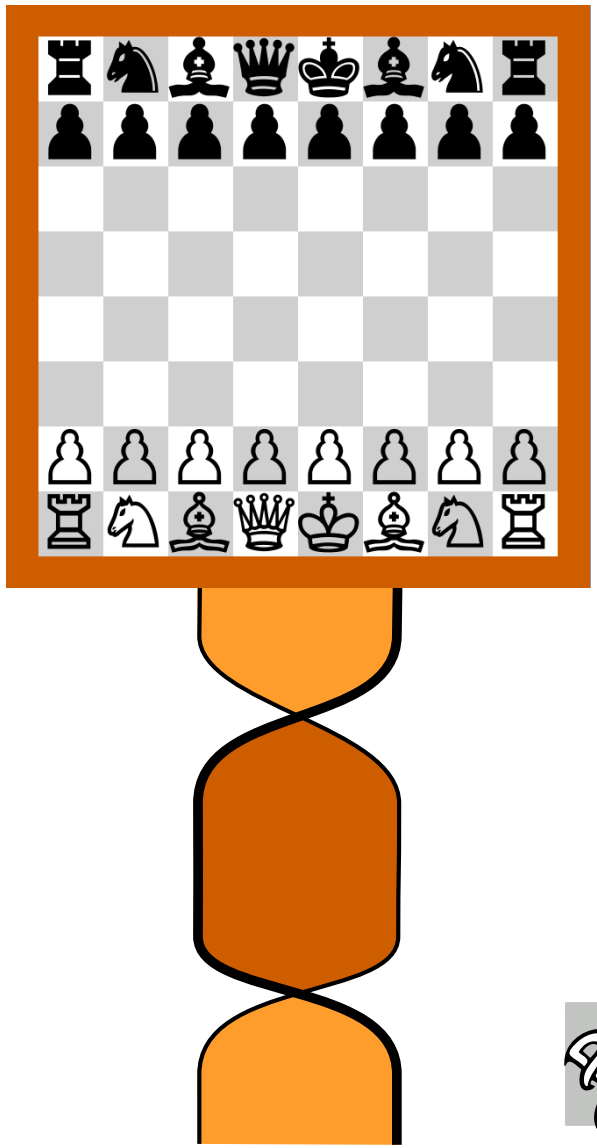
Chess game : double negation



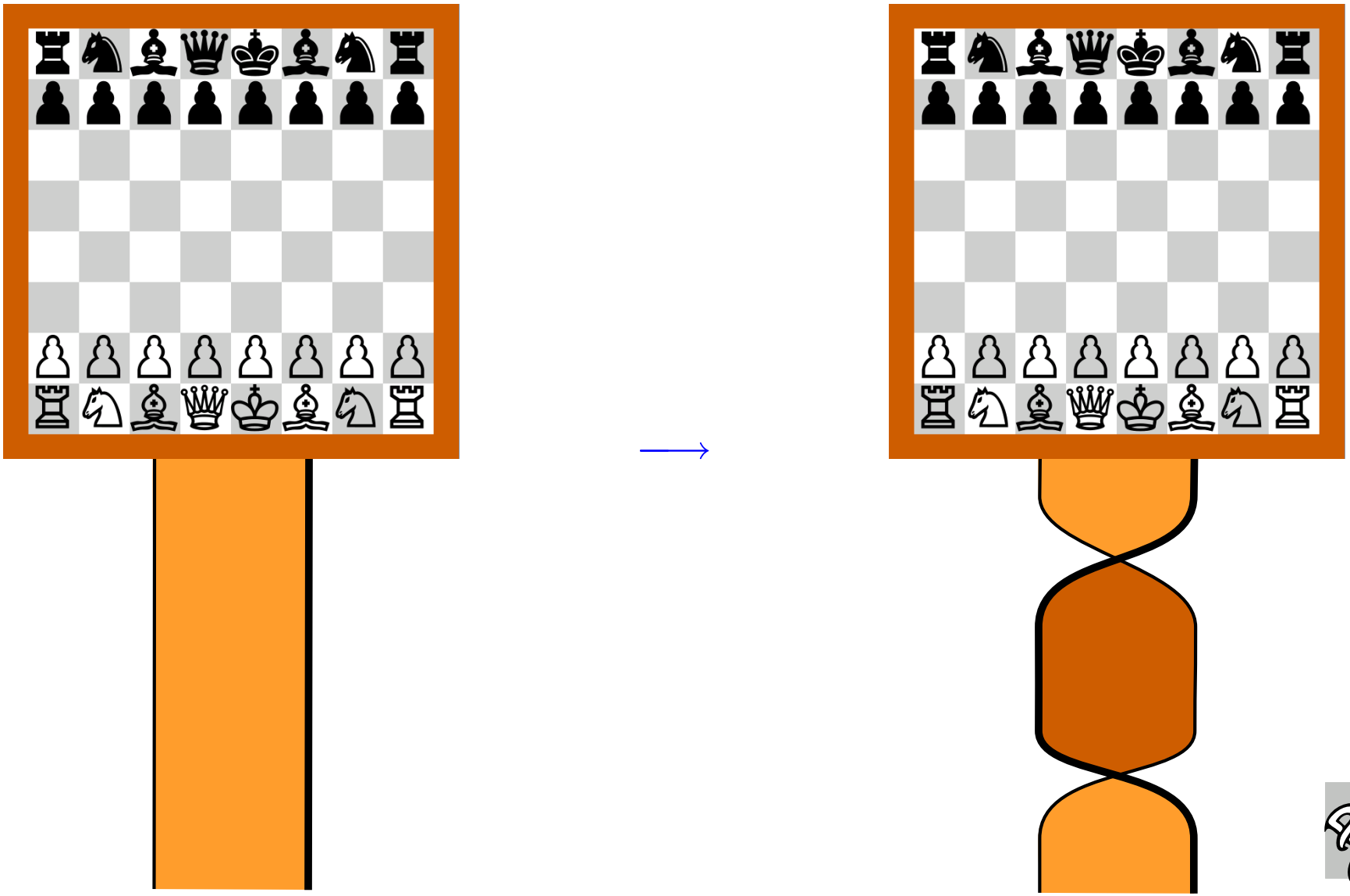
Chess game : double negation



$\neq$



Chess game : double negation



Main contributions

1. **Logical** synthesis of linear logic and game semantics : tensor logic
2. **Categorical** computation of free models of algebraic theories
3. **Algebraic** description of references using memory access modality
4. **Formalized** semantics of compilation in assembly code (Coq)

Part I

Tensor logic : a synthesis of linear logic and game semantics

Tensorial negation

Let $(C, \otimes, 1)$ be a symmetric monoidal category

Suppose that one can take the closure on the object $\bot$

$$A \multimap \bot$$

that is

$$\frac{A \otimes B \quad \to \quad C \multimap \bot}{A \quad \to \quad B \multimap (C \multimap \bot)} \quad \phi_{A,B,C}$$

Now, let us look at the functor $A \mapsto (A \multimap \bot)$

Tensorial negation

Let $(C, \otimes, 1)$ be a symmetric monoidal category

Suppose that one can take the closure on the object $\bot$

$$A \multimap \bot$$

that is

$$\frac{A \otimes B \quad \to \quad C \multimap \bot}{A \quad \to \quad (B \otimes C) \multimap \bot} \; \phi_{A,B,C}$$

Now, let us look at the functor $A \mapsto (A \multimap \bot)$

Tensorial negation

Let $(C, \otimes, 1)$ be a symmetric monoidal category

A tensorial negation is

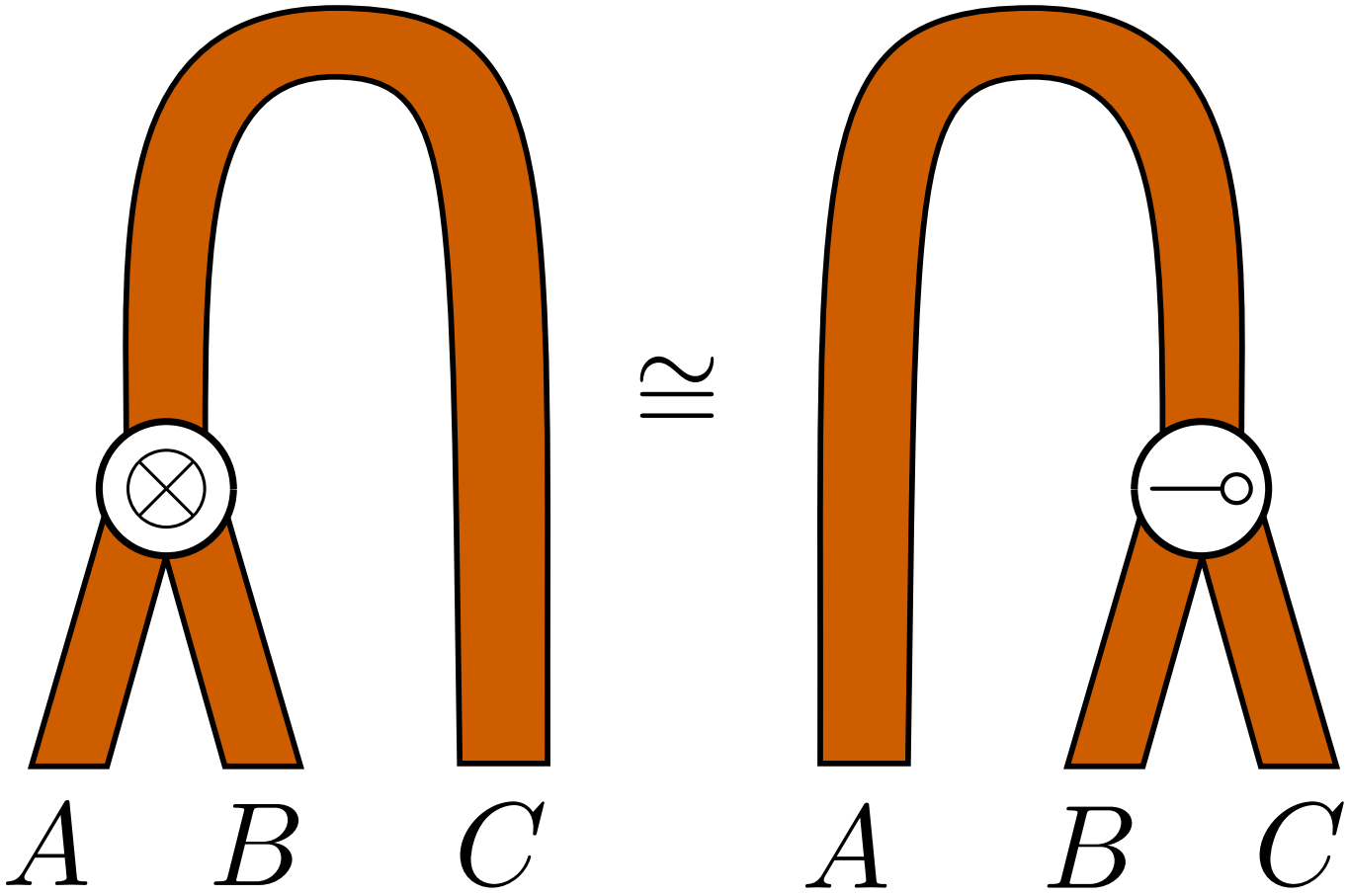
$$\neg : C \to C^{\mathrm{op}}$$

such that

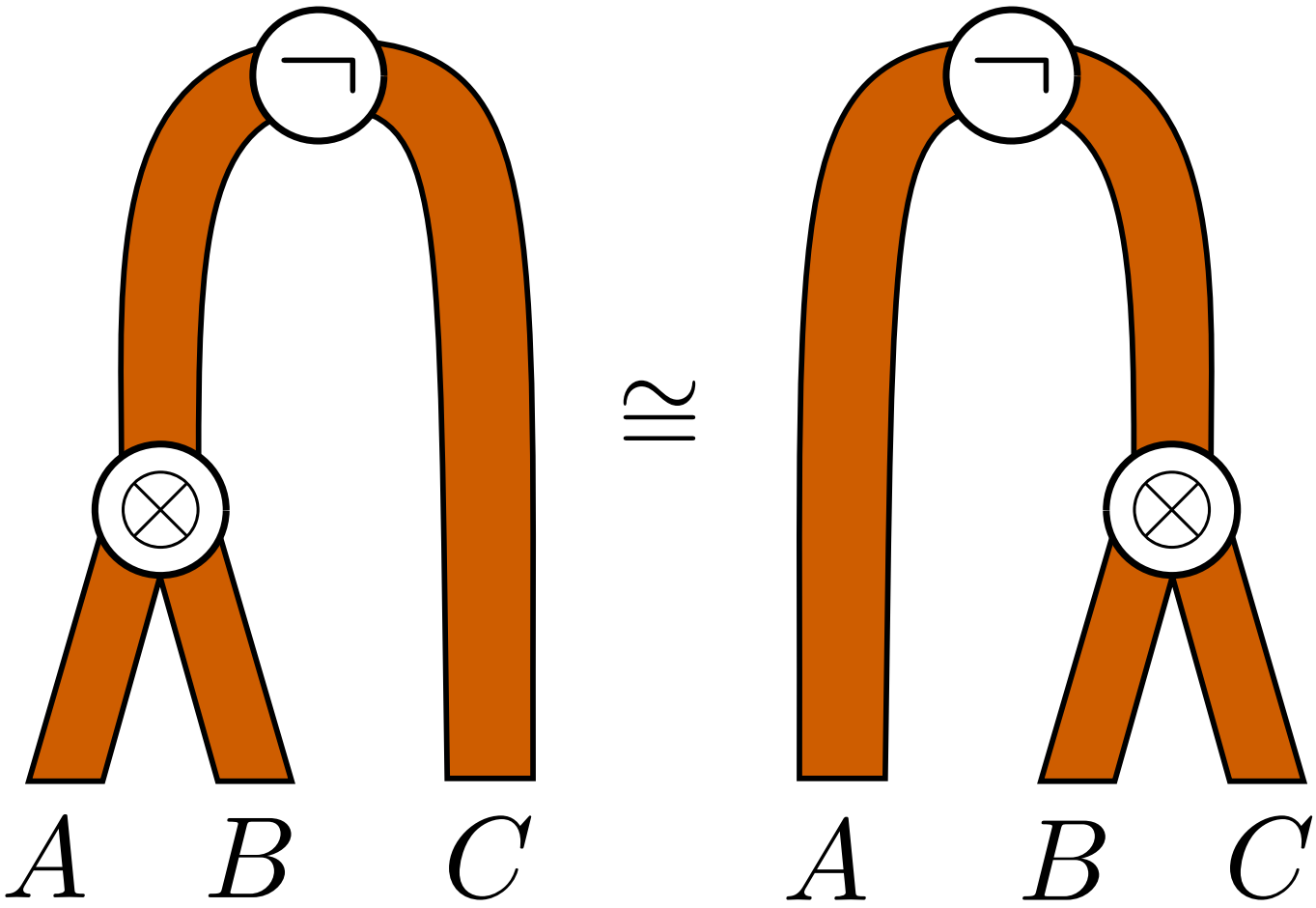
$$\frac{A \otimes B \quad \to \quad \neg C}{A \quad \to \quad \neg(B \otimes C)} \qquad \phi_{A,B,C}$$

plus a coherence diagram with the associativity of the tensor.

Closure in a symmetric monoidal category



Negation in a symmetric monoidal category



Tensor logic (multiplicative)

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} \otimes\text{-Right} \qquad \frac{\Gamma_1, A, B, \Gamma_2 \vdash [C]}{\Gamma_1, A \otimes B, \Gamma_2 \vdash [C]} \otimes\text{-Left}$$

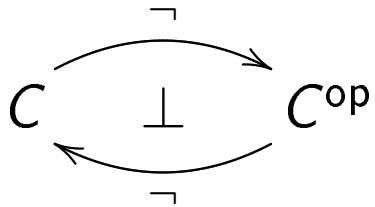
$$\frac{}{\vdash 1} \text{ 1-Right} \qquad \frac{\Gamma \vdash [A]}{\Gamma, 1 \vdash [A]} \text{ 1-Left}$$

$$\frac{\Gamma, A \vdash}{\Gamma \vdash \neg A} \neg\text{-Right} \qquad \frac{\Gamma \vdash A}{\Gamma, \neg A \vdash} \neg\text{-Left}$$

$$\frac{}{A \vdash A} \text{ Axiom} \qquad \frac{\Gamma \vdash A \qquad A, \Delta \vdash [B]}{\Gamma, \Delta \vdash [B]} \text{ Cut}$$

Self adjunction

Any tensorial negation $\neg$ induces an self-adjunction



whose monad is called the continuation monad

$$A \longrightarrow \neg\neg A$$

Continuation monad

The continuation monad $\neg\neg$ is strong

$$t_{A,B} : \quad A \otimes \neg\neg B \quad \longrightarrow \quad \neg\neg(A \otimes B)$$

with a series of coherence axioms.

Continuation monad

The continuation monad $\neg\neg$ is strong

$$t_{A,B} : \quad A \otimes \neg\neg B \quad \longrightarrow \quad \neg\neg(A \otimes B)$$

The monad is not commutative in general

Continuation monad

The continuation monad $\neg\neg$ is strong

$$t_{A,B} : \quad A \otimes \neg\neg B \quad \longrightarrow \quad \neg\neg(A \otimes B)$$

$$\neg\neg A \otimes \neg\neg B \Longrightarrow \neg\neg(A \otimes B)$$

Game semantics: left implementation

$$\neg\neg A \otimes \neg\neg B \xrightarrow{\textit{left}} \neg\neg(A \otimes B)$$

q

q

a

q

a

a

Game semantics: right implementation

$$\neg\neg A \otimes \neg\neg B \xrightarrow{\textit{right}} \neg\neg(A \otimes B)$$

q

q

a

q

a

a

Kleisli Category

We consider the Kleisli category $C_{\neg\neg}$ induced by the monad $\neg\neg$

$$\text{Objects}(C_{\neg\neg}) \quad \stackrel{\text{def}}{=} \quad \text{Objects}(C)$$

$$C_{\neg\neg}(A, B) \quad \stackrel{\text{def}}{=} \quad C(A, \neg\neg B)$$

Premonoidal structure

The kleisli category inherits a premonoidal structure.

There are two ways of forming the tensor product :

$$A \otimes B \xrightarrow{f \otimes g} \neg\neg A' \otimes \neg\neg B' \xrightarrow{left} \neg\neg(A' \otimes B')$$

$$A \otimes B \xrightarrow{f \otimes g} \neg\neg A' \otimes \neg\neg B' \xrightarrow{right} \neg\neg(A' \otimes B')$$

From tensor logic to linear logic

C_T is premonoidal iff T is strong

C_T is monoidal iff T is commutative

From tensor logic to linear logic

$C_{\neg\neg}$ is $*$-autonomous iff $\neg\neg$ is commutative

[Masahito Hasegawa, 2004]

Main novelty between tensor logic and linear logic

- **Linear logic** : the duality is unique up to isomorphism
- **Tensor logic** : one can combine several negations

Tensor logic : a more primitive logic

Tensor logic is more primitive than linear logic

Coding by Gödel translation :

tensor logic $\longrightarrow$ linear logic

intuitionistic logic $\longrightarrow$ classical logic

Tensor logic : a more primitive logic

Coding of linear logic in tensor logic

$$\vdash A_1, \ldots, A_n \quad \mapsto \quad \vdash \ \neg(\neg A_1 \otimes \cdots \otimes \neg A_n)$$

where

$$\boxed{(A_1 \parr \cdots \parr A_n) \stackrel{\text{def}}{=} \neg\,(\neg A_1 \otimes \cdots \otimes \neg A_n)}$$

generalizes the continuation monad :

$$(A) \;=\; \neg\neg A$$

Tensor logic : a polarized logic

$$\frac{\vdash \Gamma, P \qquad \vdash \Delta, Q}{\vdash \Gamma, \Delta, P \otimes Q}\otimes \qquad \frac{\vdash \Gamma_1, L, M, \Gamma_2, [P]}{\vdash \Gamma_1, L \parr M, \Gamma_2, [P]}\parr$$

$$\frac{}{\vdash 1}\,1 \qquad \frac{\vdash \Gamma, [P]}{\vdash \Gamma, \bot, [P]}\,\bot$$

$$\frac{\vdash \Gamma, L}{\vdash \Gamma, \downarrow L}\text{ Linear strengthening} \qquad \frac{\vdash \Gamma, P}{\vdash \Gamma, \uparrow P}\text{ Linear dereliction}$$

$$\frac{}{\vdash P^\bot, P}\text{ Axiom} \qquad \frac{\vdash \Gamma, P \qquad \vdash P^\bot, \Delta, [Q]}{\vdash \Gamma, \Delta, [Q]}\text{ Cut}$$

LC, Ludics [Girard], LLP [Laurent]

Coherent and finiteness spaces

Coherent and finiteness spaces
induced by the same model of tensor logic.

A model of tensor logic where negation tests the number of possible outputs of a program.

$$\text{determinism} \quad \Rightarrow \quad \text{coherent spaces}$$

$$\text{finite non determinism} \quad \Rightarrow \quad \text{finiteness spaces}$$

Category of matrices

Mat : category of matrices

- Objects : countable sets
- Morphisms : $M : X \nrightarrow Y$ is a function

$$M \quad : \quad X \times Y \to \overline{\mathbb{N}}.$$

- Identity :

$$\mathrm{id}_X \quad : \quad (x, y) \mapsto \begin{cases} 1 & \text{si } x = y \\ 0 & \text{sinon.} \end{cases}$$

- Composition :

$$N \circ M = \sum_{y \in Y} M(x, y) * N(y, z)$$

Glueing on Mat *[Hyland,Schalk]*

G(**Mat**) : the SMCC obtained by glueing along $\mathbf{Mat}(\mathbf{1}, -)$

Objects : $(X, X_\bullet)$ where $X_\bullet$ is a set of vectors of X

It can be equipped with a negation defined by focused orthogonality

scalar product : $\langle x, y \rangle \in \overline{\mathbb{N}}$

$$\neg_S(X, X_\bullet) = (X, X^\bullet) \quad \text{where} \quad X^\bullet = \{y \mid \langle x, y \rangle \in S\}$$

Fact : every $S \subset \overline{\mathbb{N}}$ induces a tensorial negation $\neg_S$

whose continuation monad is commutative.

Coherent and finiteness spaces

- $S = \{0, 1\}$:

  $\mathbf{G}(\mathbf{Mat})_{\neg_{\{0,1\}}}$ reconstructs the coherent space model.

- $S = \mathbb{N}$:

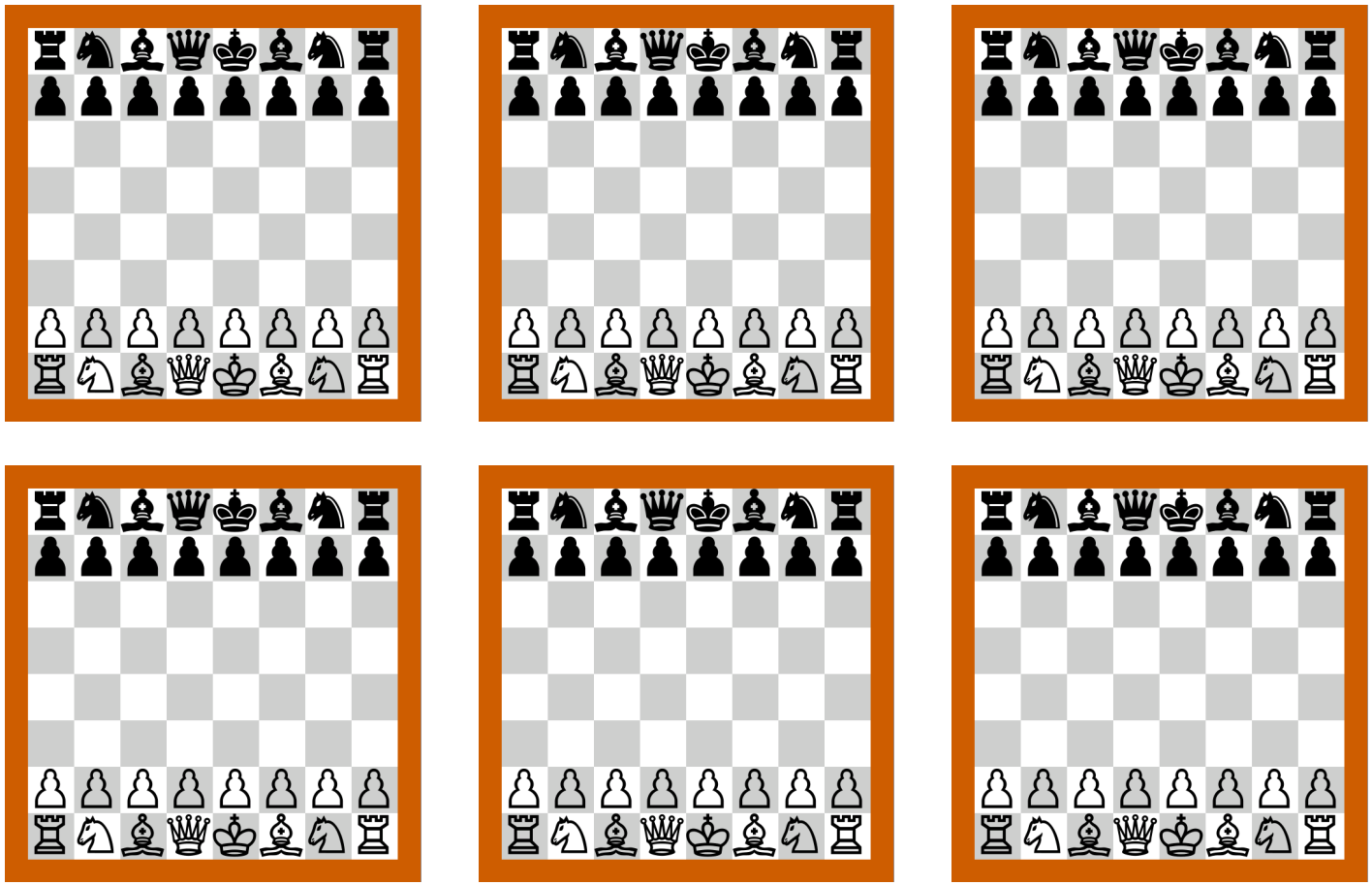
  $\mathbf{G}(\mathbf{Mat})_{\neg_{\mathbb{N}}}$ reconstructs the finiteness space model.

Resource modalities

This enables to define resource modalities in game semantics.

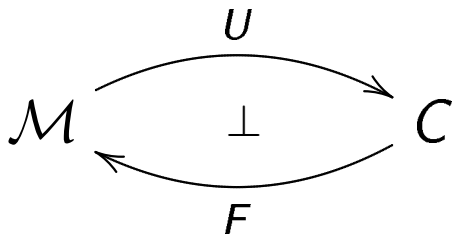
A tentative synthesis between game semantics and linear logic.

Chess game : exponential $A \mapsto !A$



Exponential modality in linear logic

A exponential modality on a $*$-autonomous category $(C, \otimes, 1)$ is defined as an adjunction :



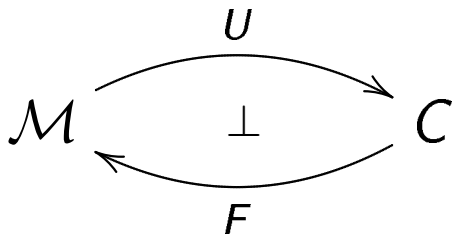
where

- $(\mathcal{M}, \times, I)$ is a cartesian category
- U is a symmetric monoidal functor

This is the usual Linear/Non-Linear model [Benton]

Exponential modality in tensor logic

A exponential modality on a symmetric monoidal category $(C, \otimes, 1)$ is defined as an adjunction :

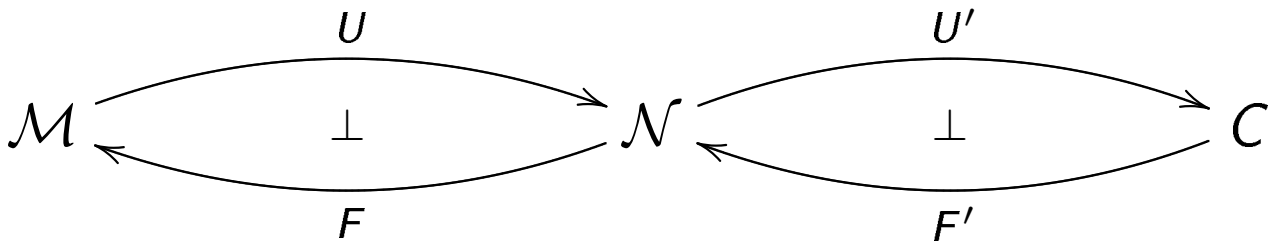


where

- $(\mathcal{M}, \times, I)$ is a cartesian category
- U is a symmetric monoidal functor

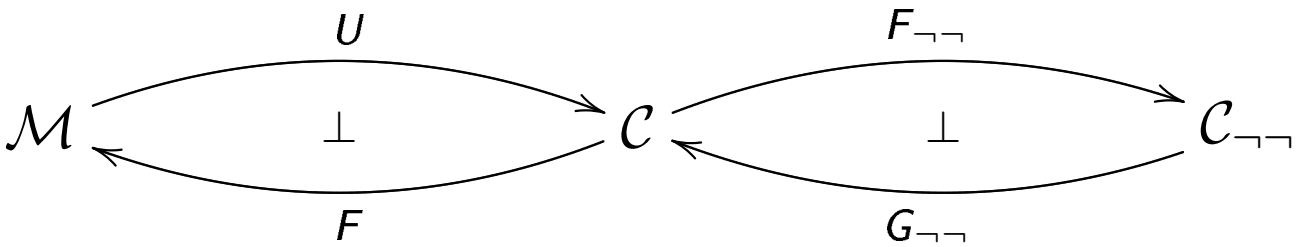
This is the same definition !

Resource modalities can be composed



From tensor logic to linear logic

When the monad $\neg\neg$ is commutative



Theorem

Any model of tensor logic (multiplicative, additive, exponential) gives rise to a model of linear logic (multiplicative, additive, exponential).

The model $\mathcal{C}_{\neg\neg}$ of linear logic is equivalent to the full subcategory of the negated objects of C.

Tensor logic: bilateral pres.

$$\frac{!\Gamma \vdash A}{!\Gamma \vdash\ !A}\ !\ \text{Strengthening} \qquad \frac{\Gamma, A \vdash}{\Gamma, !A \vdash}\ !\ \text{Dereliction}$$

$$\frac{\Gamma \vdash}{\Gamma, !A \vdash}\ !\ \text{Weakening} \qquad \frac{\Gamma, !A, !A \vdash}{\Gamma, !A \vdash}\ !\ \text{Contraction}$$

The exponential modality does not change the polarity of the formula.

Exponential in Mat

Induces the usual notion of exponentials

on coherent spaces and finiteness spaces.

Part II

Computation of free models of algebraic theories

Computing exponentials

We would like a recipe to compute

exponentials more automatically.

Computing exponentials

Observation : recursive types are not sufficiently expressive

$$!A \;=\; 1 \;\&\; (A \otimes !A)$$

replaced here by limit diagrams produced by algebra

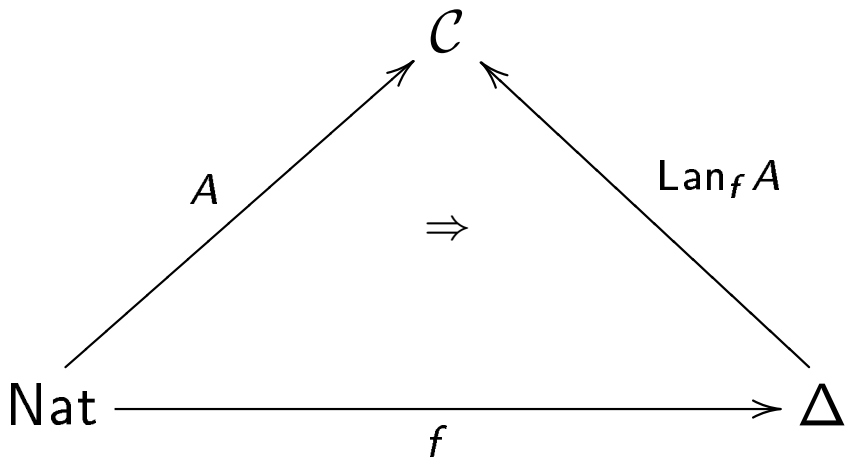
recursive formulas $\Rightarrow$ algebraic theories

Computing free monoid

A free exponential is a free commutative monoid

Idea :

- Objects of $\mathcal{C}$ $\cong$ monoidal functors from Nat to $\mathcal{C}$
- Monoids of $\mathcal{C}$ $\cong$ monoidal functors from Δ to $\mathcal{C}$
- Free monoid on A $\cong$ left Kan extension of A along Nat $\to \Delta$

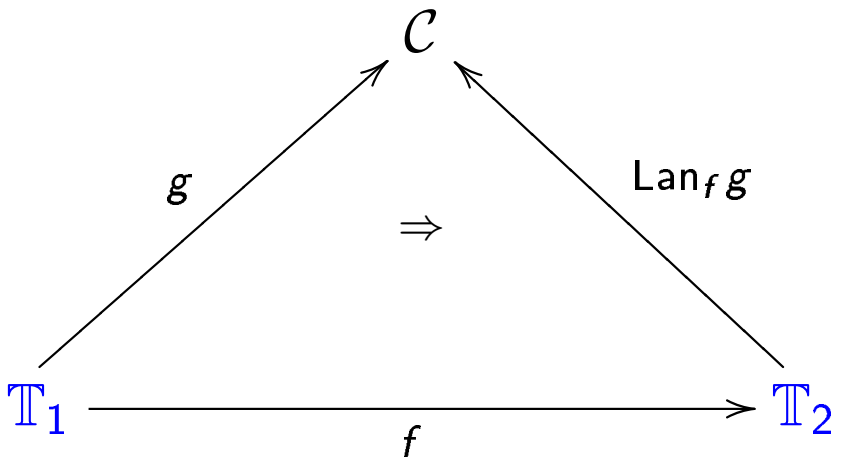


Computing free model

More generally,

to every model of a linear theory $\mathbb{T}_1$

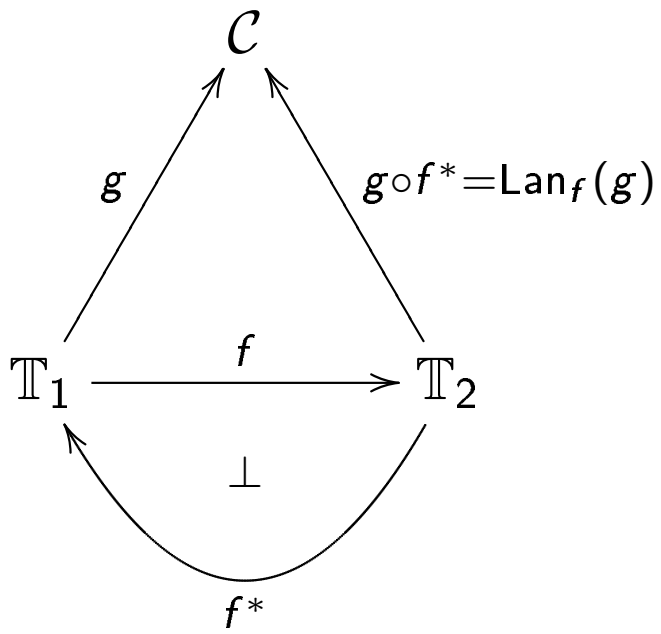
we want to associate its free $\mathbb{T}_2$-model.



Right adjoint and Kan extension

When f has a right adjoint f^*,

the left Kan extension is easy to compute



Algebraic distributors at work [Benabou]

The bicategory of distributors consists in

- Categories as 0-cells
- Functors from

$$A \times B^{\mathrm{op}} \longrightarrow \mathbf{Ens}$$

  as 1-cells, noted

$$A \nrightarrow B$$

- Natural transformations as 2-cells

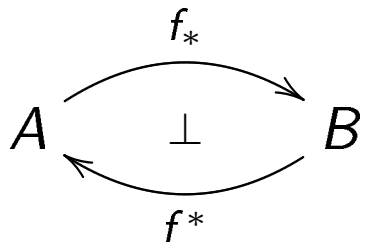
Right adjoint in Dist

Every functor $f : A \longrightarrow B$ gives rise to a distributor

$$f_* : A \nrightarrow B$$

which has a right adjoint

$$f^* : B \nrightarrow A$$

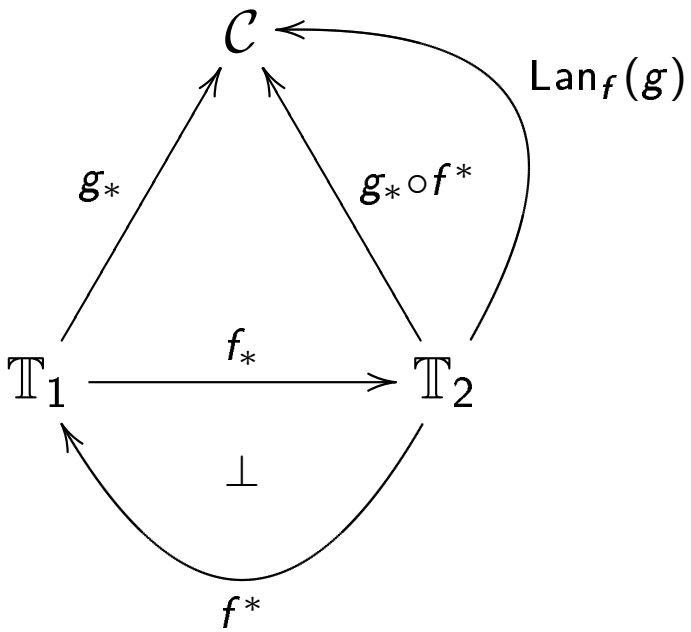


Computing Kan extension using distributors

The Kan extension of a functor f along a functor j is obtained by

- composing the distributors as $g_* \circ f^*$
- computing the functor $\mathsf{Lan}_f(g)$ that represents this distributor

$$\mathbf{Dist}(g_* \circ f^*, h_*) \cong \mathbf{Cat}(\mathsf{Lan}_f(g), h)$$



The two ingredients of the recipe

Ingredient n°1:

the adjunction
$f_* \dashv f^*$
is monoidal

$\Longrightarrow$ operadicity of f

Ingredient n°2:

the monoidal distributor
$g_* \circ f^* : A \nrightarrow C$
is represented by a monoidal functor

$\Longrightarrow$ C has the required monoidal colimits

Main result

Then, the forgetful functor from $Mon(\mathcal{C})$ to $\mathrm{Ob}(\mathcal{C})$ has a left adjoint computed by left Kan extension :

$$\mathrm{Lan}_f \quad : \quad \mathrm{Ob}(\mathcal{C}) \ \to \ Mon(\mathcal{C}).$$

This left Kan extension is given explicitly by the formula

$$\boxed{\mathrm{Lan}_f A(n) \ = \ \int^{m \in \mathrm{Nat}} \Delta(m,n) \otimes A^{\otimes m}}$$

This coend replaces the recursive formula

$$!A \ = \ 1 \ \& \ (A \otimes !A)$$

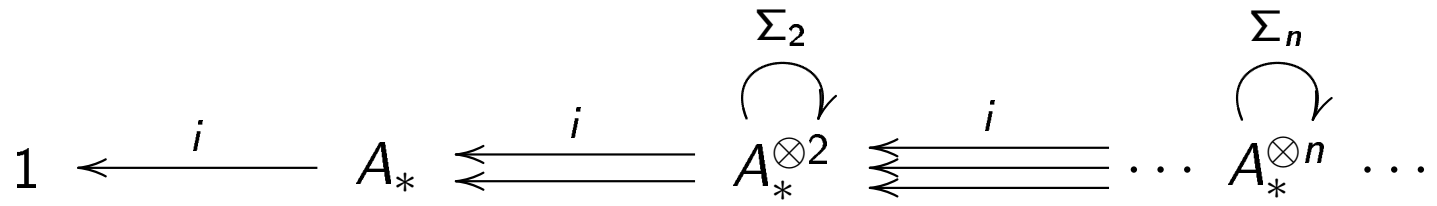
Free exponential : the recipe

Key observation : the same diagrammatic recipe computes the free modality of coherent spaces and of game semantics.

1. Take an object A of the category,
2. Compute "by hand" the free affine object

$$1 \xleftarrow[i]{} A_*$$

3. Compute the free commutative comonoid $!A$ on this pointed object, given by the limit of the diagram



Free exponential : coherent spaces and games

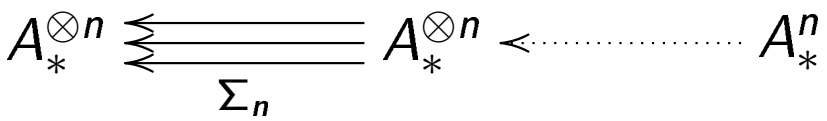
The free affine object on A

$$1 \xleftarrow{\ i\ } A_*$$

- coherent space : $A_* = A \,\&\, 1$ — A with an extra element
- Conway games : $A_* = \downarrow_w A$ — A with null payoff at the root

Free exponential : coherent spaces and games

The limit (equalizer) A^n of the diagram



- coherent space : multicliques with at most n elements of A.
- Conway games : n copies of A in sequential order

Free exponential : coherent spaces and games

The free exponential $!A$ is thus given by the limit of

$$1 \xleftarrow[i]{} A_* \xleftarrow{} A_*^2 \xleftarrow{} \cdots A_*^n \cdots$$

- coherent space : finite multicliques.
- Conway games : infinite number of copies in sequential order

The result is valid for any T-algebraic theories

- algebraic theories
- linear theories (PRO)
- symmetric theories (PROP)
- braided theories (PROB)
- projective sketches

Part III

Algebraic description of references

A study of references

One motivation for tensor logic is to clarify the algebraic structure of

general references
linear continuation and control

starting from the work of

- Abramsky, Honda and McCusker [general references]
- Parigot, Ong, Power, Robinson [$\lambda\mu$-calculus, fibered model]
- Thielecke, Selinger [control categories]
- Girard [linear logic]

A study of references

Ingredients :

- model of tensor logic
- trace operator
- memory access modality

$\Big\} \Rightarrow$ resource policy expressed as multibracketing

Decomposes in three steps the original definition of the strategy `cell`

linear cell, constant cell and memory cell

Bracketing and control

Game semantics : control is regulated by bracketing

PCF : with bracketing

$$\mathbb{B} \;=\; \neg\,\neg\,(1 \oplus 1)$$

PCF + **control** : without bracketing

$$\mathbb{B} \;=\; \neg\,!\,\neg\,(1 \oplus 1)$$

Bracketing and control

Key observation : bracketing expresses a resource policy

The resource policy of tensor logic requires to extend bracketing to multibracketing

Multibracketing

In the arena

$$(\mathbb{B} \otimes \mathbb{B}) \multimap \mathbb{B}$$

the play

$$\begin{array}{ccccc} (\mathbb{B} & \otimes & \mathbb{B}) & \multimap & \mathbb{B} \\ & & & & q_1 \\ & & q_2 & & \\ & & \mathtt{true}_2 & & \\ q_3 & & & & \\ \mathtt{true}_3 & & & & \\ & & & & \mathtt{true}_1 \end{array}$$

is well bracketed.

Multibracketing

In the arena

$$(\mathbb{B} \otimes \mathbb{B}) \multimap \mathbb{B}$$

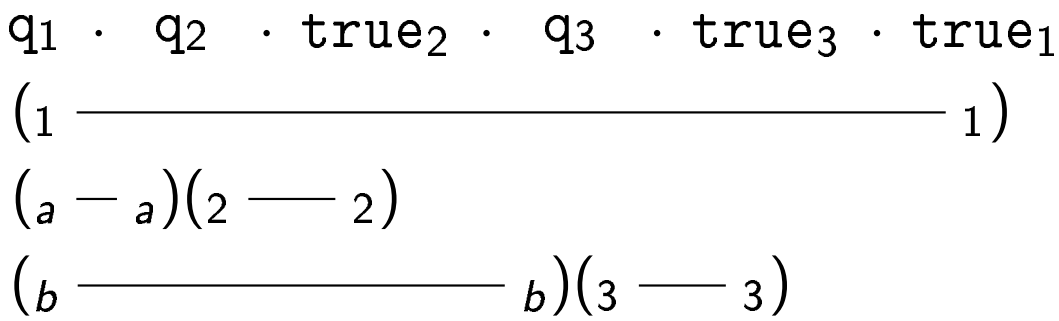
but the play

$$\begin{array}{ccccc} (\mathbb{B} & \otimes & \mathbb{B}) & \multimap & \mathbb{B} \\ & & & & q_1 \\ q_3 & & & & \\ \mathtt{true}_3 & & & & \\ & & & & \mathtt{true}_3 \end{array}$$

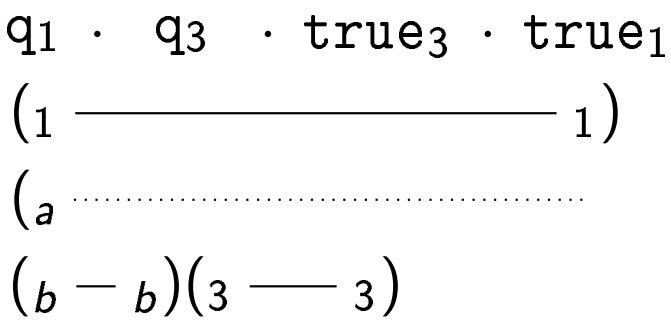
is not well bracketed.

Multibracketing

The first question emits three queries, so the play



is well bracketed, but the play



is not well bracketed because the query $(_a$ is never complied with.

Linear cell

The linear cell has type

$$(A \multimap \mathsf{Unit}) \quad \otimes \quad (\mathsf{Unit} \multimap A)$$

which we write

$$\mathtt{Write}_A \quad \otimes \quad \mathtt{Read}_A$$

Constant cell

The exponential modality

$$!A$$

provides directly a constant cell of type

$$\mathtt{Write}_A \otimes !\mathtt{Read}_A$$

Constant cell

A memory cell is a replicated constant cell up to coercion

$$!\,(\mathtt{Write}_A \otimes !\mathtt{Read}_A) \quad \longrightarrow \quad !\mathtt{Write}_A \otimes !\mathtt{Read}_A$$

Memory cell

A memory access modality is an exponential modality !

equipped with a memory access ξ,

$$\boxed{\xi_{A,B} \quad : \quad !(A \otimes !B) \longrightarrow !A \otimes !B}$$

$$\begin{array}{ccc} !(A \otimes !B) & \xrightarrow{\varepsilon_{A \otimes !B}} & A \otimes !B \\ \downarrow{\scriptstyle \xi_{A,B}} & \nearrow{\scriptstyle \varepsilon_A \otimes !B} & \\ !A \otimes !B & & \end{array} \qquad \begin{array}{ccc} !(A \otimes !B) & \xrightarrow{d_{A \otimes !B}} & !(A \otimes !B) \otimes !(A \otimes !B) \\ \downarrow{\scriptstyle \xi_{A,B}} & & \downarrow{\scriptstyle !(A \otimes \varepsilon_B) \otimes \xi_{A,B}} \\ !A \otimes !B & \xrightarrow[d_A \otimes !B]{} & !A \otimes !A \otimes !B \end{array}$$

Memory cell

The memory access modality acts as a scheduler between the different copies of A_i

$$!\mathtt{Write}_A \quad \otimes \quad !\mathtt{Read}_A$$

A soundness theorem for an arena game model of an idealized ML established by categorical means.

Part IV

Formalized semantics of compilation in assembly code (Coq)

A study of compilation

Semantic type soundness for a compiler

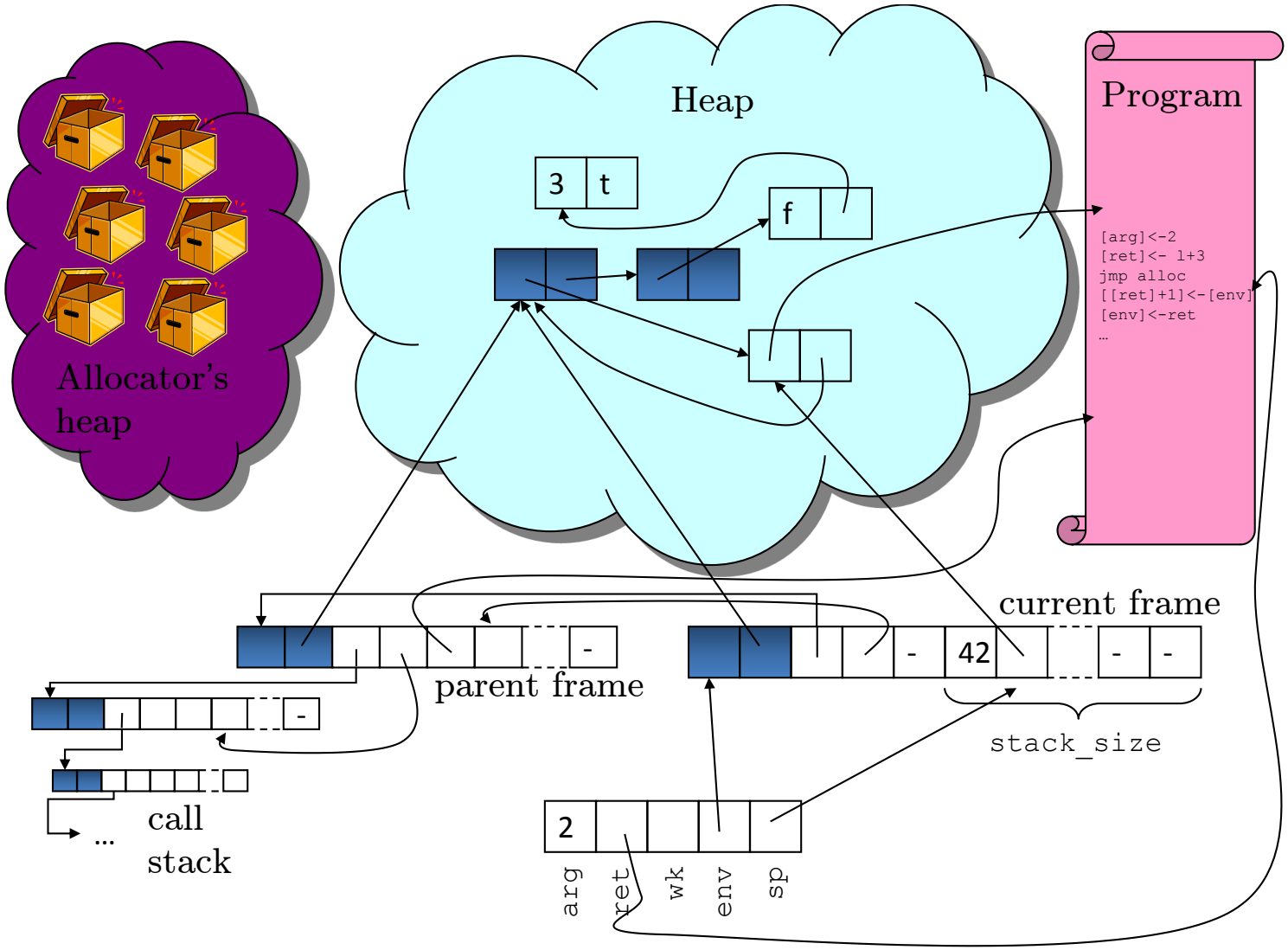
- types interpreted as relations on memory states
- enables to deal with optimized code modularly.

from a functional language with recursion into an assembly language

- specification of a function type $A \longrightarrow B$
- use of "Later" modality and Löb rule $\Rightarrow$ recursion scheme

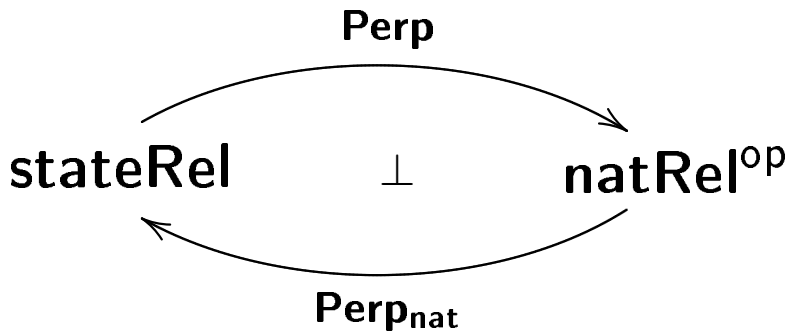
formalized in Coq

Why a Coq formalization ?



Basic ingredients

- **stateRel** : binary relation over states
- **natRel** : binary relation over labels
- an adjunction



- a separative conjunction $\otimes$, an additive conjunction $\times$
- $\Diamond$ ("Later") modality and the Löb rule

$$\frac{\Diamond\alpha \vdash \alpha}{\vdash \alpha}$$

A glance at the main theorem

```
Theorem compiler_sound :
 forall Ra init init' alloc alloc' dealloc dealloc' Γ a e (t:Γ ⊢ e : a)
    Rc Rc_cloud stack_ptr stack_ptr' n n2 n3 n' n2' n3' p p' stack_list,
 let code := compile e start alloc dealloc in
 let stack_free := extract_stacksize_from_code code in
 let code' := compile e start' alloc' dealloc' in
 let stack_free' := extract_stacksize_from_code code' in
  prog_extends_code code p start →
  prog_extends_code code' p' start' →
  AllocSpec p p' Ra init init' alloc alloc' dealloc dealloc' →
  (forall ptr ptr',
   ⊨ p p' ▷ (start+length code) (start'+length code')) :
    (memory_specification Ra Rc Rc_cloud Γ (([[a]] Ra,(ptr,ptr')) :: stack_list)
      (stack_free-1) (stack_free'-1) (stack_ptr+1) (stack_ptr'+1) n n2 n3 n' n2' n3')⊤) →
   ⊨ p p' ▷ start start' :
    (memory_specification Ra Rc Rc_cloud Γ stack_list
      stack_free stack_free' stack_ptr stack_ptr' n n2 n3 n' n2' n3')⊤.
```

Here is the theorem we have proved in Coq

A glance at the main theorem

```
Theorem compiler_sound :
 forall Ra init init' alloc alloc' dealloc dealloc' Γ a e (t : Γ ⊢ e : a)
    Rc Rc_cloud stack_ptr stack_ptr' n n2 n3 n' n2' n3' p p' stack_list,
 let code := compile e start alloc dealloc in
 let stack_free := extract_stacksize_from_code code in
 let code' := compile e start' alloc' dealloc' in
 let stack_free' := extract_stacksize_from_code code' in
  prog_extends_code code p start →
  prog_extends_code code' p' start' →
  AllocSpec p p' Ra init init' alloc alloc' dealloc dealloc' →
  (forall ptr ptr',
   ⊨ p p' ▷ (start+length code) (start'+length code')) :
    (memory_specification Ra Rc Rc_cloud Γ (([[a]] Ra,(ptr,ptr')) :: stack_list)
      (stack_free-1) (stack_free'-1) (stack_ptr+1) (stack_ptr'+1) n n2 n3 n' n2' n3')⊤) →
   ⊨ p p' ▷ start start' :
    (memory_specification Ra Rc Rc_cloud Γ stack_list
      stack_free stack_free' stack_ptr stack_ptr' n n2 n3 n' n2' n3')⊤.
```

We quantify over all pieces of code *e*

A glance at the main theorem

```
Theorem compiler_sound :
 forall Ra init init' alloc alloc' dealloc dealloc' Γ a e (t:Γ ⊢ e : a)
    Rc Rc_cloud stack_ptr stack_ptr' n n2 n3 n' n2' n3' p p' stack_list,
 let code := compile e start alloc dealloc in
 let stack_free := extract_stacksize_from_code code in
 let code' := compile e start' alloc' dealloc' in
 let stack_free' := extract_stacksize_from_code code' in
  prog_extends_code code p start →
  prog_extends_code code' p' start' →
  AllocSpec p p' Ra init init' alloc alloc' dealloc dealloc' →
  (forall ptr ptr',
   ⊨ p p' ▷ (start+length code) (start'+length code')) :
    (memory_specification Ra Rc Rc_cloud Γ (([[a]] Ra,(ptr,ptr')) :: stack_list)
      (stack_free-1) (stack_free'-1) (stack_ptr+1) (stack_ptr'+1) n n2 n3 n' n2' n3')⊤) →
   ⊨ p p' ▷ start start' :
    (memory_specification Ra Rc Rc_cloud Γ stack_list
      stack_free stack_free' stack_ptr stack_ptr' n n2 n3 n' n2' n3')⊤.
```

We compile e twice, at different places

The stack is big enough to execute the code of e

A glance at the main theorem

```
Theorem compiler_sound :
 forall Ra init init' alloc alloc' dealloc dealloc' Γ a e (t:Γ ⊢ e : a)
    Rc Rc_cloud stack_ptr stack_ptr' n n2 n3 n' n2' n3' p p' stack_list,
 let code := compile e start alloc dealloc in
 let stack_free := extract_stacksize_from_code code in
 let code' := compile e start' alloc' dealloc' in
 let stack_free' := extract_stacksize_from_code code' in
  prog_extends_code code p start →
  prog_extends_code code' p' start' →
  AllocSpec p p' Ra init init' alloc alloc' dealloc dealloc' →
  (forall ptr ptr',
   ⊨ p p' ▷ (start+length code) (start'+length code')) :
    (memory_specification Ra Rc Rc_cloud Γ (([[a]] Ra,(ptr,ptr')) :: stack_list)
      (stack_free-1) (stack_free'-1) (stack_ptr+1) (stack_ptr'+1) n n2 n3 n' n2' n3')⊤) →
   ⊨ p p' ▷ start start' :
    (memory_specification Ra Rc Rc_cloud Γ stack_list
      stack_free stack_free' stack_ptr stack_ptr' n n2 n3 n' n2' n3')⊤.
```

For any p that extends the code e compiled at start

For any p' that extends the code e compiled at start'

A glance at the main theorem

```
Theorem compiler_sound :
 forall Ra init init' alloc alloc' dealloc dealloc' Γ a e (t:Γ ⊢ e : a)
    Rc Rc_cloud stack_ptr stack_ptr' n n2 n3 n' n2' n3' p p' stack_list,
 let code := compile e start alloc dealloc in
 let stack_free := extract_stacksize_from_code code in
 let code' := compile e start' alloc' dealloc' in
 let stack_free' := extract_stacksize_from_code code' in
  prog_extends_code code p start →
  prog_extends_code code' p' start' →
  AllocSpec p p' Ra init init' alloc alloc' dealloc dealloc' →
  (forall ptr ptr',
   ⊨ p p' ▷ (start+length code) (start'+length code')) :
    (memory_specification Ra Rc Rc_cloud Γ (([[a]] Ra,(ptr,ptr')) :: stack_list)
      (stack_free-1) (stack_free'-1) (stack_ptr+1) (stack_ptr'+1) n n2 n3 n' n2' n3')⊤) →
   ⊨ p p' ▷ start start' :
    (memory_specification Ra Rc Rc_cloud Γ stack_list
      stack_free stack_free' stack_ptr stack_ptr' n n2 n3 n' n2' n3')⊤.
```

p and p' have Ra-related memory allocation routines.

A glance at the main theorem

```
Theorem compiler_sound :
 forall Ra init init' alloc alloc' dealloc dealloc' Γ a e (t:Γ ⊢ e : a)
    Rc Rc_cloud stack_ptr stack_ptr' n n2 n3 n' n2' n3' p p' stack_list,
 let code := compile e start alloc dealloc in
 let stack_free := extract_stacksize_from_code code in
 let code' := compile e start' alloc' dealloc' in
 let stack_free' := extract_stacksize_from_code code' in
  prog_extends_code code p start →
  prog_extends_code code' p' start' →
  AllocSpec p p' Ra init init' alloc alloc' dealloc dealloc' →
  (forall ptr ptr',
   ⊨ p p' ▷ (start + length code) (start' + length code') :
    (memory_specification Ra Rc Rc_cloud Γ ([[a]] Ra,(ptr,ptr')) :: stack_list)
      (stack_free-1) (stack_free'-1) (stack_ptr+1) (stack_ptr'+1) n n2 n3 n' n2' n3')⊤) →
   ⊨ p p' ▷ start start' :
    (memory_specification Ra Rc Rc_cloud Γ stack_list
      stack_free stack_free' stack_ptr stack_ptr' n n2 n3 n' n2' n3')⊤.
```

Either the two programs diverge

Or they place an element satisfying $[\![a]\!]$ at the top of the stack

Main contributions

1. **Logical** synthesis of linear logic and game semantics : tensor logic
2. **Categorical** computation of free models of algebraic theories
3. **Algebraic** description of references using memory access modality
4. **Formalized** semantics of compilation in assembly code (Coq)

game semantics $\Longleftrightarrow$ compilation

A possible synthesis ?

Merci

Outline of the manuscript

- Chapter 2
  - tensorial negation $\longrightarrow$ linear continuation monad
  - resource modalities
  - link between tensor logic and linear logic
- Chapter 3
  - free computation of resource modalities
- Chapter 4
  - Study of memory cell through the insight of tensor logic
- Chapter 5
  - study of the multicategorical structure induced by the tensorial negation
- Chapter 6
  - applied semantics :
    Typed Correctness of a compiler proved in Coq