A Computational Analysis of Logical Constants

Alberto Naibo[1] Mattia Petrolo[2] Thomas Seiller[3]

[1]Université Paris 1 - EXeCO

[2]Université Paris 7 - Università Roma Tre

[3]Institut de Mahtématiques de Luminy

August 11, 2011
Workshop on Logical Constants
ESSLLI
Ljubljana

Outline

Normalization

- Normalization plays the role of a *necessary condition* for logicality: if a connective does not satisfy normalization, then it is not a logical constant (cf. `tonk`'s problem).

- The normalization-condition does not ban all "tonkish" connectives:

- Consider the following rules

$$\ast\text{-intro}\ \frac{\Gamma \vdash A \qquad \Gamma' \vdash B}{\Gamma, \Gamma' \vdash A \ast B} \qquad\qquad \frac{\Gamma \vdash A \ast B \qquad \Gamma' \vdash A}{\Gamma, \Gamma' \vdash B}\ \ast\text{-elim}$$

- They enjoy a normalization strategy

$$\ast\text{-elim}\ \frac{\ast\text{-intro}\ \dfrac{\dfrac{\mathcal{D}}{\Gamma \vdash A} \qquad \dfrac{\mathcal{D}_1}{\Gamma' \vdash B}}{\Gamma, \Gamma' \vdash A \ast B} \qquad \dfrac{\mathcal{D}_2}{\Gamma'' \vdash A}}{\Gamma, \Gamma', \Gamma'' \vdash B} \quad \rightsquigarrow \quad \frac{\mathcal{D}_1'}{\Gamma, \Gamma', \Gamma'' \vdash B}$$

Where $\mathcal{D}_1'$ is obtained from $\mathcal{D}_1$ by adding Γ and Γ'' in the identity axioms.

Deducibility of identicals

- A connective is logical only if it uniquely (i.e. exclusively) defined by its intro/elim rules. In order to check this condition, we can check if the property of *deducibility of identicals* (Hacking 1979) is satisfied or not:

Given a n-ary connective c, the sequent $c(A_1, \ldots, A_n) \vdash c(A_1, \ldots, A_n)$ has to be derivable using only c-rules.

- Usual connectives satisfy it, e.g.

$$\dfrac{\dfrac{}{A \Rightarrow B \vdash A \Rightarrow B}\,\text{Ax} \qquad \dfrac{}{A \vdash A}\,\text{Ax}}{\dfrac{A \Rightarrow B, A \vdash B}{A \Rightarrow B \vdash A \Rightarrow B}\,\Rightarrow\text{-intro}}\,\Rightarrow\text{-elim}$$

$$\dfrac{\dfrac{\dfrac{}{A \wedge B \vdash A \wedge B}\,\text{Ax}}{A \wedge B \vdash A}\,\wedge\text{-elim}_1 \qquad \dfrac{\dfrac{}{A \wedge B \vdash A \wedge B}\,\text{Ax}}{A \wedge B \vdash B}\,\wedge\text{-elim}_2}{A \wedge B \vdash A \wedge B}\,\wedge\text{-intro}$$

- While $\ast$ does not satisfy it

$$\dfrac{\dfrac{}{A \vdash A}\,\text{Ax} \qquad \dfrac{\dfrac{}{A \ast B \vdash A \ast B}\,\text{Ax} \qquad \dfrac{}{A \vdash A}\,\text{Ax}}{A \ast B, A \vdash B}\,\ast\text{-elim}}{A, A \ast B \vdash A \ast B}\,\ast\text{-intro}$$

Towards a computational setting

- Both normalization and deducibility of identicals can be interpreted as computational operations.
- This suggests that the question of logicality can be shifted on a different level from the 'linguistic" one (sentences, propositions, etc.).
- In order to develop our proposal we have to find a suitable setting for analyzing the notion of computation. A reasonable one seems to be the λ-calculus.

Outline

Curry-Howard isomorphism

- **Set of λ-terms Λ** $\quad t := x \,|\, \lambda x.t \,|\, (t)t' \,|\, \langle t, t' \rangle \,|\, \pi_1(t) \,|\, \pi_2(t)$
- λ-terms t are considered as programs; a type judgement $t : A$ is a program equipped with a specification (i.e. a proposition) describing its behavior.
- β-reduction corresponds to the execution of a program t, when applied in a certain context $\mathcal{C}(\,)$; β-reduction shows how t computes.
- **Set of evaluation contexts Σ** $\quad \mathcal{C}(\,) := (\,)t_1 \dots t_n \,|\, \pi_1(\,) \,|\, \pi_2(\,)$
- The Curry-Howard isomorphism establishes a one-to-one correspondance between natural deduction and λ-calculus, e.g.

$$\dfrac{\dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \Rightarrow B}\,\Rightarrow\text{-intro} \qquad \Gamma' \vdash u : A}{\Gamma, \Gamma' \vdash (\lambda x.t)u : B}\,\Rightarrow\text{-elim} \quad \rightsquigarrow \quad \Gamma, \Gamma' \vdash t[^{u}/_{x}] : B$$

$$\dfrac{\dfrac{\Gamma \vdash t : A \qquad \Gamma' \vdash u : B}{\Gamma, \Gamma' \vdash \langle t, u \rangle : A \wedge B}\,\wedge\text{-intro}}{\Gamma, \Gamma' \vdash \pi_1(\langle t, u \rangle) : A}\,\wedge\text{-elim} \quad \rightsquigarrow \quad \Gamma \vdash t : A$$

Indeed, *normalization* corresponds to β-reduction.

η-expansion and deducibility of identicals

- Deducibility of identicals exactly corresponds to η-expansion:

$$\dfrac{\dfrac{\dfrac{}{t : A \Rightarrow B \vdash t : A \Rightarrow B}\,\text{Ax} \qquad \dfrac{}{x : A \vdash x : A}\,\text{Ax}}{t : A \Rightarrow B, x : A \vdash (t)x : B}\,\Rightarrow\text{-elim}}{t : A \Rightarrow B \vdash \lambda x(t)x : A \Rightarrow B}\,\Rightarrow\text{-intro}$$

$$\dfrac{\dfrac{\dfrac{}{t : A \wedge B \vdash t : A \wedge B}\,\text{Ax}}{t : A \wedge B \vdash \pi_1(t) : A}\,\wedge\text{-elim}_1 \qquad \dfrac{\dfrac{}{t : A \wedge B \vdash t : A \wedge B}\,\text{Ax}}{t : A \wedge B \vdash \pi_2(t) : B}\,\wedge\text{-elim}_2}{t : A \wedge B \vdash \langle \pi_1(t), \pi_2(t) \rangle : A \wedge B}\,\wedge\text{-intro}$$

- In λ-calculus the main objects are programs, which are *intensional objects*.
- In order to work in an extensional setting, the following rules (η-*expansion*) are needed:

$$t \longrightarrow_\eta \lambda x(t)x$$

(with $x \notin FV(t)$)

$$t \longrightarrow_\eta \langle \pi_1(t), \pi_2(t) \rangle$$

Maximality of $\equiv_{\beta\eta}$

- We can define $\beta\eta$-equivalence ($\equiv_{\beta\eta}$) as the smallest equivalence relation containing $\longrightarrow_\beta$ and $\longrightarrow_\eta$.
- Can we add some other equivalence relations on λ-terms?
- No. A consequence of **Böhm's theorem** is that

Any equivalence relation strictly extending $\equiv_{\beta\eta}$ forces the identification of all λ-terms.

- This result suggests to consider normalization and deducibility of identicals as the only two necessary conditions for logicality.
- Starting from pure λ-terms, is it possible to define logical constants by using operations based on $\equiv_{\beta\eta}$?

A computational definition of types

- $\mathcal{SN} :=$ the set of strongly $\beta\eta$-normalizable λ-terms.

Definition 2.1 (Orthogonality)

$t \perp\!\!\!\perp \mathcal{C}(\,)$ if and only if $\mathcal{C}(t) \in \mathcal{SN}$.

- $A \subset \Lambda$ orthogonal to $E \subset \Sigma$ means that the objects of A behave in the same manner when *tested* with the elements of E, hence E defines a specification/type of the elements of A.

A computational definition of types

- A set $A \subset \Lambda$ corresponds to a type if and only if there exists a set $E \subset \Sigma$, such that $A = E^{\perp}$. This is equivalent to

Definition 2.2 (Type)

$A \subset \Lambda$ is a type if and only if $A = A^{\perp\perp}$.

- The two following sets are types

$$\{t \,|\, \forall u \in A, (t)u \in B\}^{\perp\perp} = A \Rightarrow B$$

$$\{t \,|\, \pi_1(t) \in A, \pi_2(t) \in B\}^{\perp\perp} = A \wedge B$$

It can be shown that the closure by bi-orthogonal is not necessary (*internal completeness*).

Thesis

Operations on λ-terms generating types are *logical* operations.

Problem: *petitio principii*

- In reconstructing logical types by starting from pure λ-terms an error of *petitio principii* has been committed: we implicitly assume what we want to justify as logical.

1. The form of pure λ-terms reflects the form of already given typed inference rules.

2. The notion of evaluation context presupposes that we already know the inferential behavior of the types we want to define.

- The problem can be rephrased in the following manner: how can we define new pure λ-terms and operations on them, without passing through types in advance?

Problem: stability of reduction

- In λ-calculus the notion of reduction (normalization/execution) is not stable: the introduction of new constructions on lambda-terms imposes a modification of the notion of reduction, which is necessarily guided by an operation on primitive types (i.e. an operation on propositions).

- If we don't want to pass through a primitive notion of type we have to work in a framework where reduction is defined as primitive and in a unique manner, and where the distinction between logical and non logical operations can be based on this notion of reduction.

- In order to do it the basic idea is to change the set of proof-objects considered.

Outline

Overview

- The computational level is taken as primitive.
- Our leading idea is that the basic computational properties (of programs) are:

  1. Composition / Execution;
  2. Termination.

- Given a sets of "objects" (mathematical objects), a notion of execution and a notion of termination allows one to construct types, as in lambda-calculus, but with the possibility of having a *unique* and *stable* notion of composition/execution.
- This type of investigation is inspired by the so-called *geometrical characterization of proofs*. E.g. proof-nets, ludics, geometry of interaction.

Framework	**Execution**	**Termination**	**Proof-terms**
λ-calculus	β-reduction	(Strong) Normalizability	Syntactical objects
Permutations	Paths	No "internal" cycles	Mathematical objects
Ludics	Normalization	Daemon	
GoI	Execution	Nilpotency/Convergency	

A toy example: permutations

- λ-terms represent proofs by "codifying" the order of the rules used in proofs.
- If we look instead at the geometrical structure of proofs, we can use other proof-objects, as generalizations of permutations (e.g. in GoI one uses partial isometries acting on a Hilbert space) or trees (as in ludics).
- Let us consider a toy example

Definition 1 (Untyped proof)

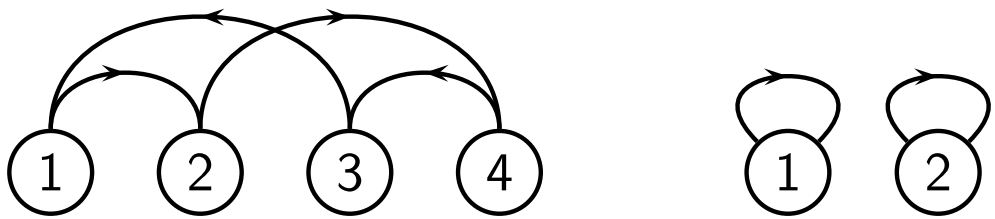
An untyped proof *is a pair* $\mathfrak{a} = \langle X, \sigma \rangle$, *where:*

1. $X \in \wp_f(\mathbb{N}) \setminus \{\emptyset\}$ *is called the* location *of* $\mathfrak{a}$;
2. σ *is a permutation on* X.

Composition

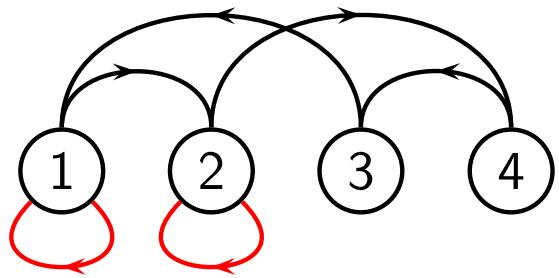
- Like programs, permutations can be composed.
- Let's consider two untyped proofs:

  for exemple, $\mathfrak{a} = \langle\{1,2,3,4\},(1,2,4,3)\rangle$ and $\mathfrak{b} = \langle\{1,2\}, id\rangle$:



Composition

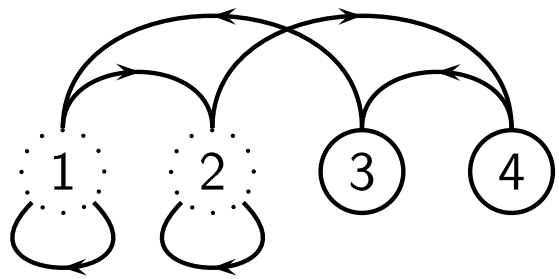
- Their composition is obtained by plugging them together:



- Correspondence with the operation of application in pure λ-calculus.

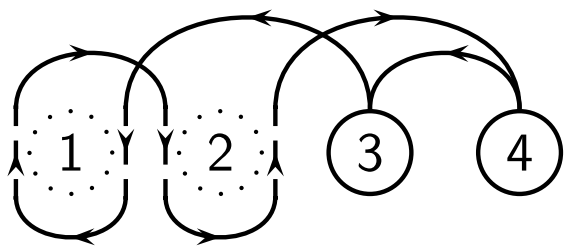
Execution

- The execution of this composition gives as result:



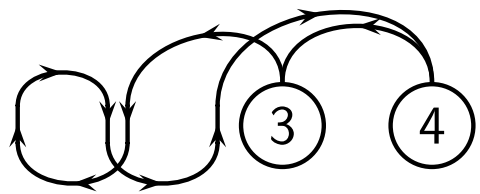
Execution

- The execution of this composition gives as result:



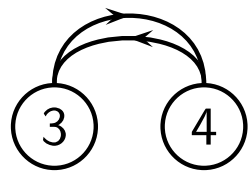
Execution

- The execution of this composition gives as result:



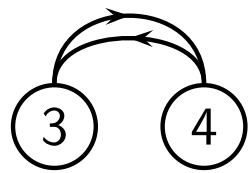
Execution

- The execution of this composition gives as result:



Execution

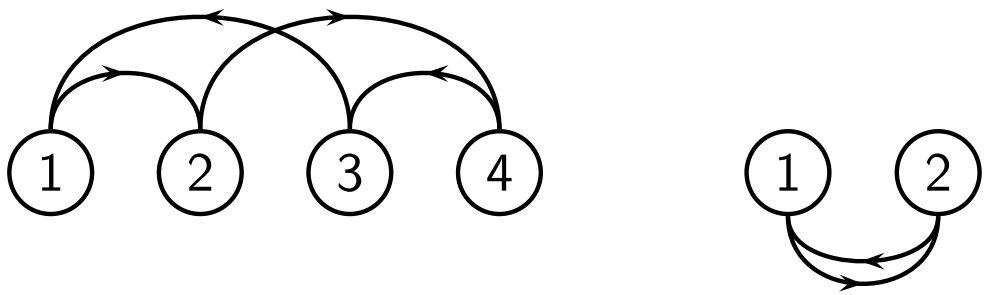
- The execution of this composition gives as result:



- Execution corresponds to β-reduction in λ-calculus.

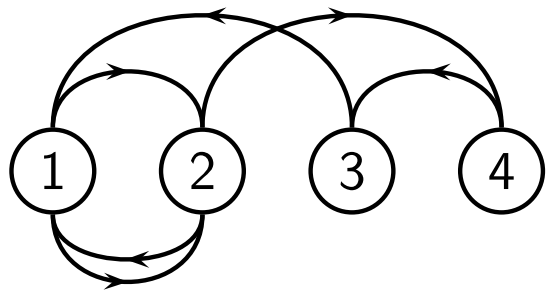
Looping executions

- Sometimes the composition of untyped proofs can generate "internal" cycles (loops).
- For example, let $\mathfrak{a} = \langle \{1, 2, 3, 4\}, (1, 2, 4, 3) \rangle$ and $\mathfrak{b}' = \langle \{1, 2\}, (1, 2) \rangle$:



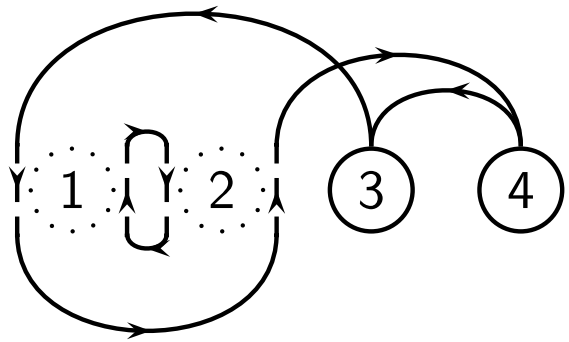
Looping executions

- The execution of the composition brings to:



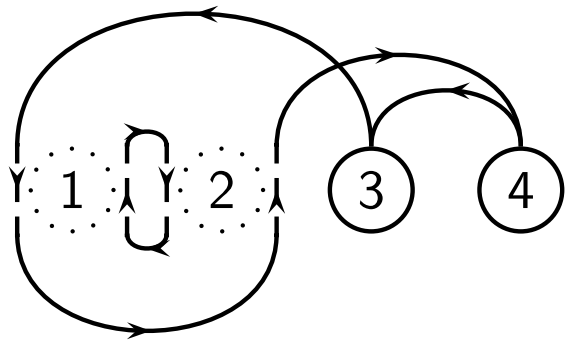
Looping executions

- The execution of the composition brings to:



Looping executions

- The execution of the composition brings to:



- The presence of internal cycles means that the computation does not terminate.

- There is an analogy with non-terminating reductions of pure λ-terms, e.g. $(\lambda x(x)x)\lambda x(x)x$.

Terminating executions

- In order to work uniquely with genuine computational objects, we have to consider only terminating executions.

Definition 2 (Termination)

The execution of a composition terminates if there is no internal cycles.

- This restriction is similar to the one done on the class of pure λ-terms so to obtain just (strongly) normalizable terms.

- However, there is a fundamental difference:
  - In the case of λ-calculus the restriction is done by selecting a subclass of "good" λ-terms through the use of an *a priori* given set of types;
  - In our case the restriction is done directly on the notion of composition without passing through any "ready-made" notion of type.

Application of untyped proofs

Definition 3 (Application)

Let be $\mathfrak{a} = \langle X \cup Y, \sigma \rangle$ *and* $\mathfrak{b} = \langle Y, \tau \rangle$*, with* $X \cap Y = \emptyset$ *and let* π_X *be the partial identity on* X*.*
The application of $\mathfrak{a}$ *to* $\mathfrak{b}$ *is defined when no internal cycles appear and it is then defined as:*

$$[\mathfrak{a}]\mathfrak{b} = \langle X, \sigma \dagger \tau \rangle$$

where

$$\sigma \dagger \tau = \pi_X(\sigma \cup \sigma\tau\sigma \cup \sigma\tau\sigma\tau\sigma \ldots)\pi_X$$

Orthogonality and types

- Analogously to λ-calculus we have

Definition 4 (Orthogonality)

Two untyped proofs $\mathfrak{a} = \langle X, \sigma \rangle$ *and* $\mathfrak{b} = \langle X, \tau \rangle$ *are* orthogonal ($\mathfrak{a} \perp\!\!\!\perp \mathfrak{b}$) *if and only if* $\sigma\tau$ *is a cyclic permutation.*

- **Note**. The composition of two orthogonal untyped proofs represents a special case of a terminating application.

Definition 5 (Type)

A subset $\mathbf{A}$ *of* $\mathfrak{S}(X)$ *equal to its bi-orthogonal* $\mathbf{A}^{\perp\perp}$ *is called a* type *(of carrier* X*).*

- **Remark 1**. An untyped proof can belong to different types.
- **Remark 2**. This is not an inductive notion of type: we don't have any notion of atomic type.

Logical and non-logical operations

- Every operation on untyped proofs allows one to define an operation on types. For instance, if $g(\mathfrak{x}, \mathfrak{y})$ defines an untyped proof from two untyped proofs $\mathfrak{x}$ and $\mathfrak{y}$, we define the operation on types

$$g(A, B) = \{g(\mathfrak{x}, \mathfrak{y}) \,|\, \mathfrak{x} \in A, \mathfrak{y} \in B\}^{\perp\perp}$$

- Given a type it is also always possible to define its dual at the level of types. However, this dual cannot always be expressed through an operation over untyped proofs.

Logicality condition (a proposal)

A type is *logical* when both it and its dual have a computational definition, i.e. when they are both defined as operations on untyped proofs.

Logical operations

- With our toy example it is possible to define multiplicative connectives of linear logic.
- Let $\mathfrak{a} = \langle X, \sigma \rangle$ and $\mathfrak{b} = \langle Y, \tau \rangle$, where $X \cap Y = \emptyset$. We can define the *tensor product* of $\mathfrak{a}$ and $\mathfrak{b}$ by:

$$\mathfrak{a} \otimes \mathfrak{b} = \langle X \cup Y, \sigma \cup \tau \rangle$$

-

Logical operations

- The operation of application induces a type.
- Let be $\mathbf{A}$ and $\mathbf{B}$ two types and consider the set

$$\mathbf{A} \multimap \mathbf{B} = \{\mathfrak{f} \mid \forall \mathfrak{a} \in \mathbf{A}, [\mathfrak{f}]\mathfrak{a} \in \mathbf{B}\}$$

Theorem 1

The following equivalence holds: $\mathbf{A} \multimap \mathbf{B} = (\mathbf{A} \otimes \mathbf{B}^{\perp})^{\perp}$.

Corollary 1

The set $\mathbf{A} \multimap \mathbf{B}$ *is a type.*

-

Logical operations: a further example

- Define $\mathscr{T}(\mathbf{A}) = \{\langle X, \tau^{-1}\rangle \mid \langle X, \tau\rangle \in \mathbf{A}\}^{\perp\perp}$.
- Then $(\mathscr{T}(\mathbf{A}))^{\perp} = \mathscr{T}(\mathbf{A}^{\perp})$.
- This construction on types satisfies a lot of good properties:
  - It preserves correctness: the inverse of a disjoint union of transpositions is a disjoint union of transpositions;
  - It is "natural": if $\sigma \subset \tau$, then $\sigma^{-1} \subset \tau^{-1}$;
  - It is "internally complete": $\mathscr{T}(\mathbf{A}) = \{\langle X, \tau^{-1}\rangle \mid \langle X, \tau\rangle \in \mathbf{A}\}$.
- The operator $\mathscr{T}$ can be associated to the following rules:

$$\frac{\vdash A}{\vdash \mathscr{T}(A)}\ \mathscr{T}\text{-intro} \qquad \frac{\vdash \mathscr{T}(A)}{\vdash A}\ \mathscr{T}\text{-elim}$$

Non-logical operations: an example

- Let's take an untyped proof $\mathfrak{a} = \langle X, \sigma \rangle$ and define the operation of *square* exponentiation:
$$\mathfrak{a}^2 = \langle X, \sigma^2 \rangle$$
- Given a type $\mathbf{A}$, the square operation over untyped proofs induces the new type:
$$\boxdot\mathbf{A} = \{\mathfrak{a}^2 \mid \mathfrak{a} \in A\}^{\perp\perp}$$

Take a look at some examples:

- The sets $\mathbf{F}_2 = \{\langle\{1,2\}, id\rangle\}$ and $\mathbf{F}_3 = \{\langle\{1,2,3\}, id\rangle\}$ are types.
- We have
$$\mathbf{C}_2 = \{\langle\{1,2\},(1,2)\rangle\} = \mathbf{F}_2^{\perp}$$
$$\mathbf{C}_3 = \{\langle\{1,2,3\},(1,2,3)\rangle, \langle\{1,2,3\},(1,3,2)\rangle\} = \mathbf{F}_3^{\perp}$$
- These equalities are satisfied: $\boxdot\mathbf{F}_2 = \mathbf{F}_2$, $\boxdot\mathbf{C}_2 = \mathbf{F}_2$, $\boxdot\mathbf{F}_3 = \mathbf{F}_3$, $\boxdot\mathbf{C}_3 = \mathbf{C}_3$, hence $(\boxdot\mathbf{C}_2)^{\perp} = \mathbf{C}_2$ and $(\boxdot\mathbf{C}_3)^{\perp} = \mathbf{F}_3$

Non-logical operations: an exemple

- There is not a general and univocal operation over permutations that permits to define $(\boxdot\mathbf{A})^{\curlywedge}$, for *every* type $\mathbf{A}$.
- This operation should be an operation f on permutations, such that if $\mathfrak{b} = \langle X, \tau \rangle \in \mathbf{A}^{\curlywedge}$, then $\langle X, f(\tau) \rangle \in (\boxdot\mathbf{A})^{\curlywedge}$.
- This means that f should respect the following condition: for every $\mathfrak{a} = \langle X, \sigma \rangle \in \mathbf{A}$ and $\mathfrak{b} = \langle X, \tau \rangle \in \mathbf{A}^{\curlywedge}$, if $\tau\sigma$ is cyclic then $f(\tau)\sigma^2$ is cyclic.
- It is not possible to define a unique f for every type $\mathbf{A}$.

By way of conclusion: advantages of untyped frameworks

- The logicality condition we imposed is not an absolute one.
- The untyped framework is rich enough for entailing a *pluralist* perspective: each operation on untyped proofs defines an operation on types and there already exists many properties of untyped proofs and types that could be used to distinguish between logical and non logical operations on types, for instance:
  - the preservation of correctness, i.e. given an untyped proof $\mathfrak{a} = \langle X, \sigma \rangle$, $\mathfrak{a}$ has to be a disjoint union of transpositions, namely $\sigma^2 = Id$ and $\sigma(x) \neq x$ for all $x \in X$;
  - the "naturality", i.e. the preservation of inclusion in case of permutations;
  - internal completeness, i.e. the closure by bi-orthogonality is not necessary;
  - ...

Some references

- J.-Y. Girard 2001. Locus solum. *Mathematical Structures in Computer Science*, 11 (3): 301–506.
- J.-Y. Girard 2003. From foundations to ludics. *Bulletin of Symbolic Logic*, 9 (2): 131–168.
- J.-Y. Girard 2011. Geometry of interaction V: Logic in the hyperfinite factor. *Theoretical Computer Science*, 412 (20): 1860–1883.
- I. Hacking 1979. What is logic?. *Journal of Philosophy*, 76 (6): 285–319.
- D. Prawitz 1971. Ideas and results in proof theory. In *Proceedings of the 2nd Scandinavian Logic Symposium*, ed. J. Fenstad, 237-309, Amsterdam: North-Holland.
- C. Riba 2009. On the Values of Reducibility Candidates. *TLCA 2009*: 264–278.
- T. Seiller. Graphs of interaction: Multiplicatives, under revision for publication in *Annals of Pure and Applied Logic*.