

# High-level signatures and initial semantics

Ambroise Lafont

joint work with Benedikt Ahrens, André Hirschowitz, Marco Maggesi

CSL 2018

# Introduction

We are interested in signatures specifying (untyped) languages with variables and substitution (e.g. lambda-calculus).

A family of lists of natural numbers yields a **combinatorial signature**:

- each list specifies an operation in the language.
- the number of its arguments is the size of the list
- each natural number in the list indicates the number of bound variables in the corresponding argument

**Example : Lambda-calculus:**

application (2 arguments)

lambda-abstraction (1 argument  
binding 1 variable)

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

# High-level signatures

Endofunctors may be considered as signatures, generalizing combinatorial signatures.

There is a natural **category of models** of such a signature: the category of its algebras.

The specified language (or **syntax**) is characterized as the initial object in the category of models.

This definition is motivated by the **recursion principle** induced by the initiality property.

# Purpose of our work

The specified language (or **syntax**) is characterized as the initial object in the category of models.



The initial object may not exist and therefore the signature does not specify anything!

**Goal of our work:** Identify a large class of high-level signatures (they are not exactly endofunctors in our setting) that actually specify a language.

# Table of contents

## **1. Languages, monads and modules**

## 2. Signatures and their models

## 3. Recursion

## 4. Presentables signatures

# Monads

A monad **R** corresponds to a language with variables as placeholders for any expression of **R**.

**R(X)** denotes the set of expressions taking variables in **X**.  
Intuitively, it should contain at least the set **X** of variables.

Given any family  $(\mathbf{t}_x)_{x \in \mathbf{X}}$  of elements of **R(Y)**, any expression **e** in **R(X)** can be substituted to yield an expression **e**[**x**  $\mapsto$  **t<sub>x</sub>**] in **R(Y)**.

The substitution is required to satisfy some intuitive equations.

# Operations as module morphisms

In the lambda-calculus,

$$\text{app}(t, u)[x \mapsto v_x] = \text{app}(t[x \mapsto v_x], u[x \mapsto v_x])$$

i.e.

**application commutes with substitution**

Let us rewrite the right hand side:

$$\text{app}(t, u)[x \mapsto v_x] = \text{app}((t, u)[x \mapsto v_x])$$

considering the obvious substitution on pairs of lambda terms.

We say that pairs of lambda-terms form a **module** over the lambda-calculus monad, and application is a **module morphism**.

# Module over a monad

A module **M** over a monad **R** corresponds to expressions with variables as placeholders for any expression in the language **R**.

Given a module **M**, the set **M(X)** is the set of expressions taking variables in **X** (but contrary to monads, a variable may not immediately yield a generalized expression).

Given any family  $(t_x)_{x \in X}$  of expressions in **R(Y)**, any expression **e** in **M(X)** can be substituted to yield an expression **e**[**x**  $\mapsto$  **t<sub>x</sub>**] in **M(Y)**.

As for monads, the substitution is required to satisfy some intuitive equations.



# Examples of modules

## Modules over a monad:

Some examples of modules over a monad **R**:

- **R** itself
- **R x R** (i.e. pairs of expressions of **R**)
- **M x N** for any module **M** and **N**

## Important example : Derivative of a module

- **R'** is the module defined by  $\mathbf{R}'(\mathbf{X}) = \mathbf{R}(\mathbf{X} + \{\mathbf{x}\})$  for any set **X** of variables
- more generally, we similarly define **M'** given a module **M**

The new variable **x** is used to model an operation binding a variable (e.g. the lambda-abstraction)

# Examples of module morphisms

A module morphism between two modules  $M$  and  $N$  on the same monad  $R$  is a family of maps  $(f_x: M(X) \rightarrow N(X))_x$  commuting with substitution.

## Examples:

$$id_M : M \rightarrow M$$

the family of identity maps  $(id_{M(X)}: M(X) \rightarrow M(X))_X$  for any module  $\mathbf{M}$

$$app : L \times L \rightarrow L$$

the application operation of the lambda calculus monad  $\mathbf{L}$ .

## Binding variables:

In  $\lambda x.t$ , the term  $t$  depends on an additional free variable  $x$ :

If  $\lambda x.t \in L(Y)$ , then  $t \in L(Y + \{x\}) = \mathbf{L}'(\mathbf{Y})$

**abs:  $\mathbf{L}' \rightarrow \mathbf{L}$**  is a module morphism

# Table of contents

1. Languages, monads and modules

**2. Signatures and their models**

3. Recursion

4. Presentables signatures

# Signatures

A **signature**  $\Sigma$  assigns (functorially) to each monad  $R$  a module  $\Sigma_R$  over it.

A **model** of a signature  $\Sigma$  is a monad  $R$  together with a morphism of modules  $\sigma_R : \Sigma_R \rightarrow R$ .

Models form a category (morphisms are monad morphisms commuting with  $\sigma$ ).

The **syntax generated by** a signature  $\Sigma$  is the initial object in its category of models.

This notion of signature is too general in the sense that we do not expect that this initial object always exists.

# Examples of syntax generating signatures:

-  $R \mapsto R \times R$

models are monads  $R$  that comes with a module morphism  $R \times R \rightarrow R$ .  
The syntax corresponds to a language with variables and a binary operator **b**:

$$\begin{aligned} \text{expr} ::= & x && (\text{variable}) \\ & | \mathbf{b}(t, u) && \text{where } t \text{ and } u \text{ are any expressions} \end{aligned}$$

-  $R \mapsto R \times R + R'$

By universal property of the disjoint sum  $+$ , models are monads  $R$  equipped with two module morphisms  $R \times R \rightarrow R$  and  $R' \rightarrow R$ .  
The syntax corresponds to lambda calculus.

# Algebraic signatures

More generally, any signature of the form  $R \mapsto R' \times R'' \times R''' + R \times R'' \times R''' \times R + \dots$  (i.e. any disjoint sum of products of finite derivatives of the monad) generates a syntax.

These **algebraic signatures** correspond to low-level signatures. They specify languages with  $n$ -ary operations binding a finite number of variables in their arguments.

**Our main result:** quotients of algebraic signatures also generate a syntax

# Table of contents

1. Languages, monads and modules
2. Signatures and their models
3. Recursion
- 4. Presentables signatures**

# Quotient of a signature

## Quotient of a set:

A quotient of a set  $X$  is a set  $Y$  together with a surjection  $p : X \rightarrow Y$ .

$$x \sim x' \iff p(x) = p(x')$$

## Quotient of a signature:

A quotient of a signature  $\Sigma$  is a signature  $\Psi$  together with a (natural) family of module morphisms  $(f_R : \Sigma_R \rightarrow \Psi_R)_R$  that is pointwise surjective.

A **presentable signature** is a quotient of an algebraic signature.

**Main Theorem:** Any presentable signature generates a syntax.



# Examples of presentable signatures

Presentable signatures allow to extend syntax generated by algebraic (or low-level) signatures with new kinds of operations.

## **A binary commutative operation:**

as a quotient of the signature of a binary operation  $R \mapsto R \times R$  by the action of the symmetry.

## **Syntactic closure operator:**

TODO

# FIN PROVISOIRE

Ne pas lire les slides qui suivent (ce sont des anciennes slides que je garde au cas où).

# FIN PROVISOIRE

Ne pas lire les slides qui suivent (ce sont des anciennes slides que je garde au cas où).

# Examples of monads (à supprimer ?)

- the syntax of arithmetic expressions
- the (untyped) syntax of lambda-calculus  $L$  (*modulo alpha equivalence*)

$\text{expr} ::= x$	<i>(variable)</i>
$\quad   t\ u$	<i>(application)</i>
$\quad   \lambda x.t$	<i>(abstraction)</i>

- the (untyped) syntax of lambda-calculus modulo beta-equivalence and eta-equivalence

# 'High-level' VS classical signatures

+ Our 'high-level' signatures are more abstract and contrast with 'low-level' signatures which seem quite ad-hoc.

— Our signatures, are too general: **we don't expect that all of them specify a language** (i.e. that the initial object always exist in the category of models associated to a signature).

## Goal of our work:

Identify a large class of (high-level) signatures which actually specify a language.

# Combinatorial signatures

A **combinatorial signature** is given by a family of lists of natural numbers:

- each list specifies an operation in the language.
- the number of its arguments is the size of the list
- each natural number in the list indicates the number of bound variables in the corresponding argument

## Example : Lambda-calculus:

Two operations:

application (2 arguments)

lambda-abstraction (1 argument  
binding 1 variable)

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

# Copie de Languages as monads

## A monad **A** as a language with variables:

- for each set  $X$ , a set  $A(X)$  of expressions taking free variables in  $X$ .
- any variable  $x \in X$  is a valid expression that we note  $\text{var}_X(x) = \underline{x} \in A(X)$
- given a family  $(t_x)_{x \in X}$  of expressions in  $A(Y)$ , we can perform for any expression **e** in **A(X)** the substitution  $e[x \mapsto t_x]$  lying in  $A(Y)$

Three monadic laws:

$$\text{COMPOSITION OF SUBSTITUTIONS} \quad e[x \mapsto t_x][y \mapsto u_y] = e[x \mapsto t_x[y \mapsto u_y]]$$

$$\text{IDENTITY SUBSTITUTION} \quad e[x \mapsto x] = e$$

$$\text{VARIABLE SUBSTITUTION} \quad \forall x \in X \quad x[y \mapsto t_y] = t_x$$

# Overview of the methodology

1. Introduce a notion of signature.
2. Construct an associated notion of model (suitable as domain of interpretation of the syntax generated by the signature). Such models form a category.
3. Define the syntax generated by a signature as its initial model, when it exists.
4. Identify a class of signatures that generate a syntax: **presentable signatures**



# Copie de Operations as module morphisms

:



For each set  $X$ , the sum of two expressions  $e, e' \in A(X)$  take free variables in  $X$ :

$$\begin{aligned} \forall X, \text{ add}_X : A(X) \times A(X) &\rightarrow A(X) \\ (e, e') &\mapsto e + e' \end{aligned}$$

Note that (*commutation with substitution*):

$$(e + e')[x \mapsto t_x] = e[x \mapsto t_x] + e'[x \mapsto t_x]$$

We characterize this situation as follows:

$A(X) \times A(X)$  expressions are "*substitutable*"   $A \times A$  is a **module** on  $A$   
 $\text{add}$  commutes with substitution   $\text{add}$  is a **module morphism**

# Examples of monads

- the assignement  $X \mapsto \mathcal{P}(X) = \{ U \mid U \subset X \}$  yields a monad  $\mathcal{P}$ .

$$\forall X, \text{var}_X : X \rightarrow \mathcal{P}(X)$$
$$x \mapsto \{x\}$$

Let  $U \subset X$  (i.e.  $U \in \mathcal{P}(X)$ ) and  $(V_x)_{x \in X}$  a family of subsets of  $Y$ .

Substitution is defined as union:

$$U[x \mapsto V_x] = \bigcup_{x \in U} V_x \in \mathcal{P}(Y)$$

# Induction

## Example: computing the free variables of a lambda-term

We compute it by induction on the syntax:

$$fv(x) = \{x\} \quad \text{(variable)}$$

$$fv(tu) = fv(t) \cup fv(u) \quad \text{(application)}$$

$$fv(\lambda x.t) = fv(t) \setminus \{x\} \quad \text{(abstraction)}$$

This is formalized in our setting as a family of maps  $(fv_x: L(X) \rightarrow \mathcal{P}(X))_x$  which *commutes with variable and substitution*:

$$\begin{aligned} fv(var_L(x)) &= \{x\} \\ &= var_{\mathcal{P}}(x) \end{aligned} \qquad \begin{aligned} fv(u[x \mapsto t_x]_L) &= \bigcup_{y \in fv(u)} t_y \\ &= fv(u)[x \mapsto fv(t_x)]_{\mathcal{P}} \end{aligned}$$

(This is a definition of a monad morphism)

# Induction

## Example: computing the free variables of a lambda-term

*fv* also commutes with 'application' and 'abstraction'

$$\begin{aligned} app_{\mathcal{P}} : \mathcal{P} \times \mathcal{P} &\rightarrow \mathcal{P} \\ (V, V') &\mapsto V \cup V' \end{aligned}$$

$$\begin{aligned} abs_{\mathcal{P}, X} : \overbrace{\mathcal{P}'(X)}^{\mathcal{P}(X + \{n\})} &\rightarrow \mathcal{P} \\ V &\mapsto V \setminus \{n\} \end{aligned}$$

Actually, these commutations **define** *fv* uniquely by induction:

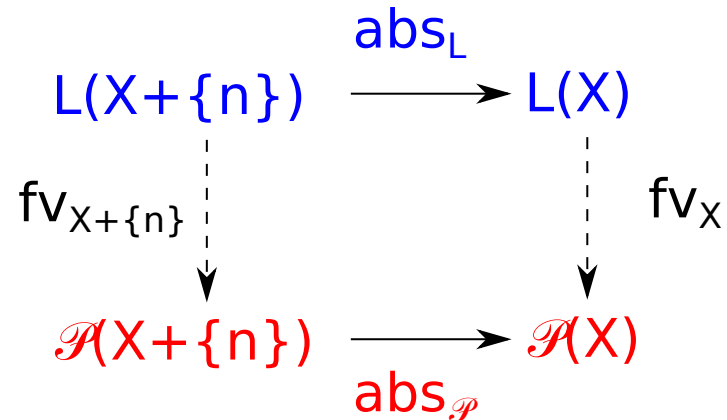
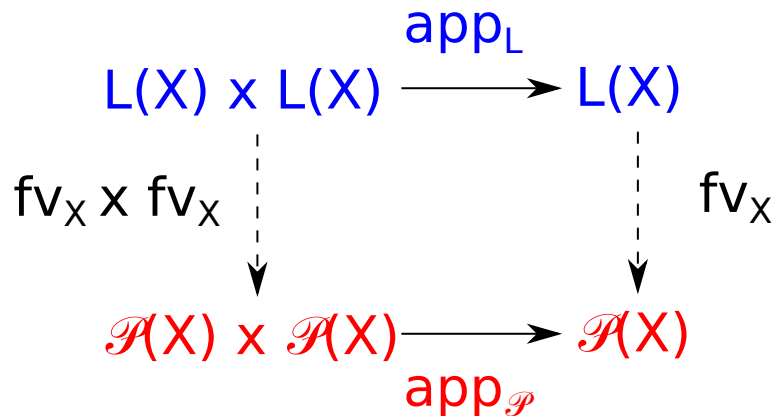
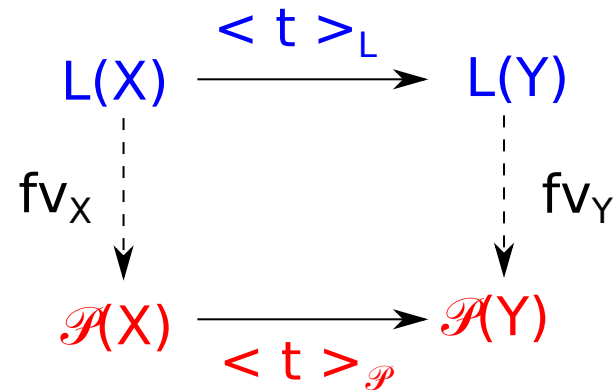
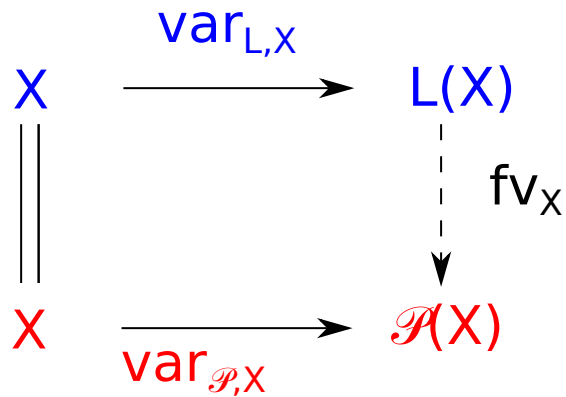
$$fv(x) = \{x\} \quad \text{(commutation with variable)}$$

$$fv(tu) = fv(t) \cup fv(u) \quad \text{(commutation with application)}$$

$$fv(\lambda x.t) = fv(t) \setminus \{x\} \quad \text{(commutation with abstraction)}$$

# Induction and initiality

$fv$  is the unique family of maps that makes the following diagrams commute:



# Induction and initiality

More generally, let  $R$  be a monad with application and abstraction.

$$X \xrightarrow{\text{var}_{R,X}} R(X)$$

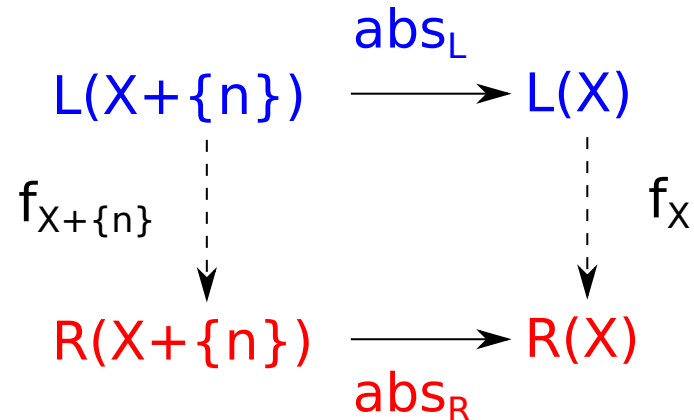
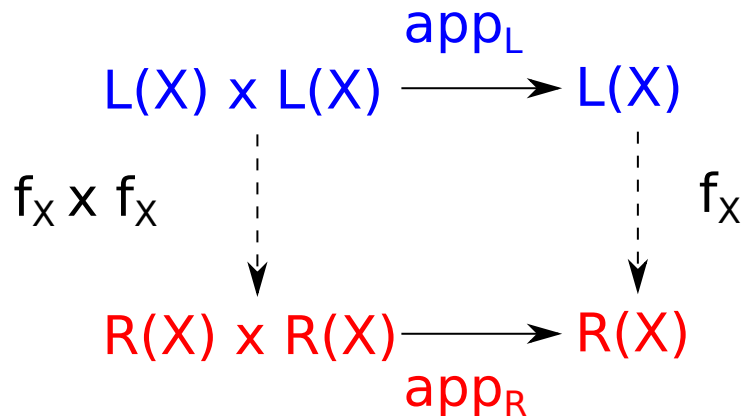
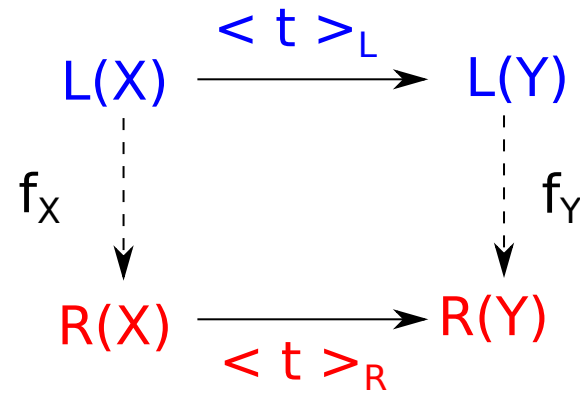
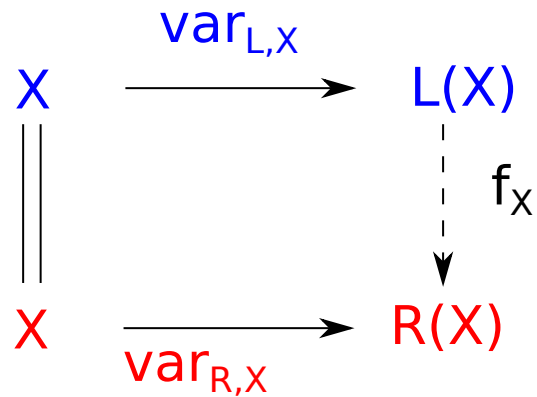
$$R(X) \xrightarrow{\langle t \rangle_R} R(Y)$$

$$R(X) \times R(X) \xrightarrow{\text{app}_R} R(X)$$

$$R(X + \{n\}) \xrightarrow{\text{abs}_R} R(X)$$

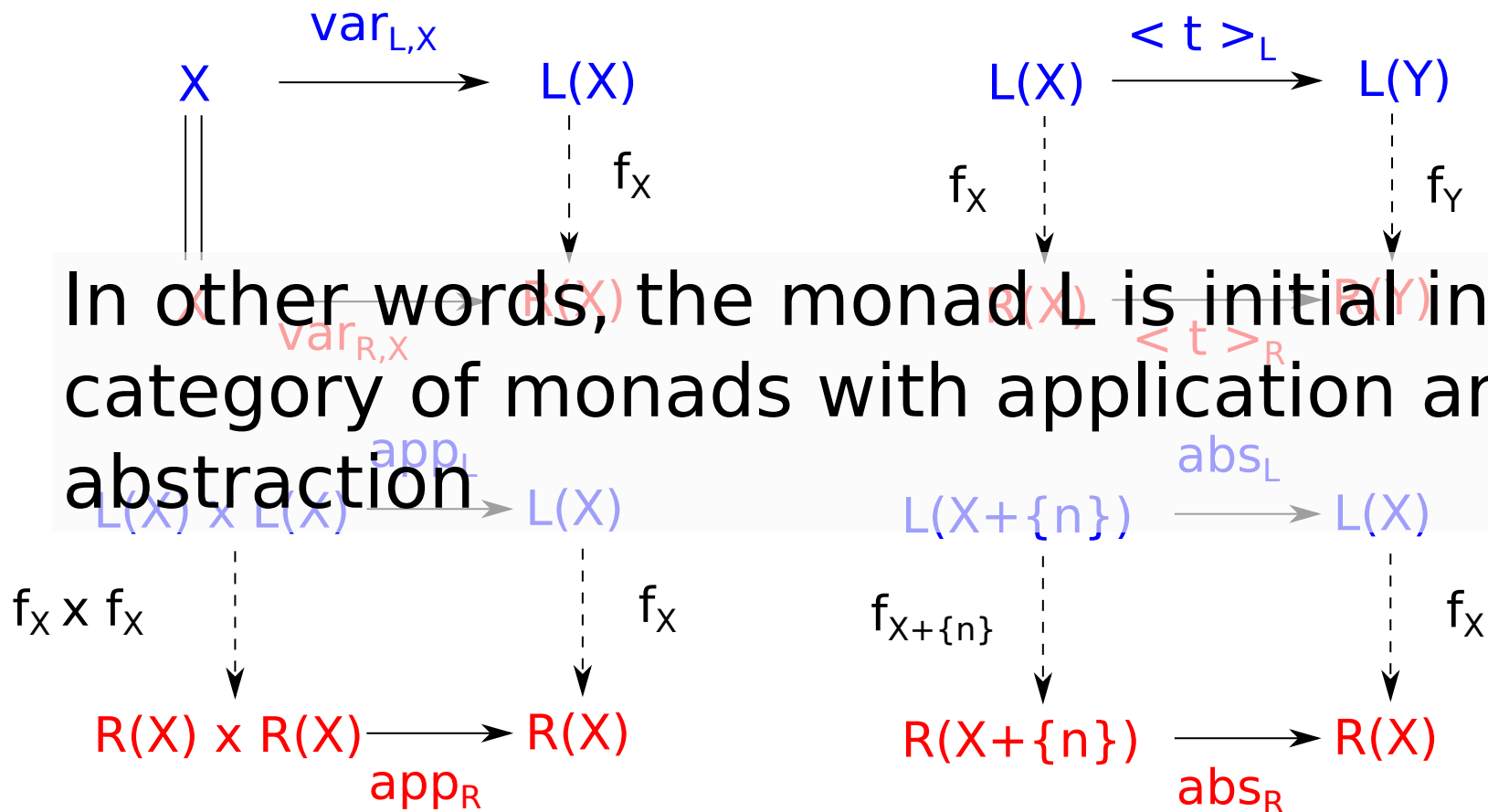
# Induction and initiality

More generally, let  $R$  be a monad with application and abstraction. Then there is a unique family  $(f_x)_x$  of maps (defined by induction) that makes the following diagrams commute:



# Induction and initiality

More generally, let  $R$  be a monad with application and abstraction. Then there is a unique family  $(\mathbf{f}_X)_X$  of maps (defined by induction) that makes the following diagrams commute:





# Syntax and initiality

## A definition of a syntax:

A **syntax** is a monad that comes with an *induction principle*, i.e. which is initial in a suitable category of *monads + operations that it implements*.

## Example:

The monad L of lambda calculus is initial in the category of *monads + application and abstraction*.


We say that L is the **syntax generated by the signature of application and abstraction**.

We will now present a general definition of **signatures**.

# Signatures

## What a signature should be:

$L$  is initial among the monads  $R$  that model the signature  $\Sigma_L$  of application and abstraction, i.e. monads  $R$  that come with module morphisms:

$$\begin{array}{l} app_R : R \times R \rightarrow R \\ abs_R : R' \rightarrow R \end{array} \quad \text{or} \quad [app_R, abs_R] : \underbrace{R \times R + R'}_{\Sigma_L(R)} \rightarrow R$$


A syntax  $S$  is initial among the monads  $R$  that model its associated signature  $\Sigma$ , i.e. monads  $R$  that come with a module morphism:

$$\sigma_R : \Sigma_R \rightarrow R$$

Thus, a signature  $\Sigma$  should assign to any monad  $R$  a module  $\Sigma_R$  over it.

# Signatures

Let  $\mathbf{R}$  be a monad that models the signature of application and abstraction. Then there exists a unique monad morphism  $\mathbf{f} : \mathbf{L} \rightarrow \mathbf{R}$  which commutes with abstraction and application:

$$\begin{array}{ccc}
 L(X) \times L(X) & \xrightarrow{\text{app}_L} & L(X) \\
 \downarrow \mathbf{f}_X \times \mathbf{f}_X & & \downarrow \mathbf{f}_X \\
 R(X) \times R(X) & \xrightarrow{\text{app}_R} & R(X)
 \end{array}$$

$$f_X(\text{app}_L(t, u)) = \text{app}_R(f_X(t), f_X(u))$$



(and similarly for abs)

$$f_X(\text{abs}_L(t)) = \text{abs}_R(f_{X+\{n\}}(t))$$

Let  $\mathbf{R}$  be a monad that models a signature  $\Sigma$  (there is a module morphism  $\sigma_R : \Sigma_R \rightarrow \mathbf{R}$ ). Then there exists a unique monad morphism  $\mathbf{f} : \mathbf{S} \rightarrow \mathbf{R}$  which commutes with  $\sigma$ :

$$\begin{array}{ccc}
 \Sigma_L(X) & \xrightarrow{\sigma_L} & L(X) \\
 \downarrow \text{??} & & \downarrow \mathbf{f}_X \\
 \Sigma_R(X) & \xrightarrow{\sigma_R} & R(X)
 \end{array}$$

# Signatures

Let  $\mathbf{R}$  be a monad that models the signature of application and abstraction. Then there exists a unique monad morphism  $\mathbf{f} : \mathbf{L} \rightarrow \mathbf{R}$  which commutes with abstraction and application:

$$\begin{array}{ccc}
 L(X) \times L(X) & \xrightarrow{\text{app}_L} & L(X) \\
 \downarrow \mathbf{f}_X \times \mathbf{f}_X & & \downarrow \mathbf{f}_X \\
 R(X) \times R(X) & \xrightarrow{\text{app}_R} & R(X)
 \end{array}$$

$$f_X(\text{app}_L(t, u)) = \text{app}_R(f_X(t), f_X(u))$$



(and similarly for abs)

$$f_X(\text{abs}_L(t)) = \text{abs}_R(f_{X+\{n\}}(t))$$

Let  $\mathbf{R}$  be a monad that models a signature  $\Sigma$  (there is a module morphism  $\sigma_R : \Sigma_R \rightarrow \mathbf{R}$ ). Then there exists a unique monad morphism  $\mathbf{f} : \mathbf{S} \rightarrow \mathbf{R}$  which commutes with  $\sigma$ :

$$\begin{array}{ccc}
 \Sigma_L(X) & \xrightarrow{\sigma_L} & L(X) \\
 \downarrow \Sigma(\mathbf{f})_X & & \downarrow \mathbf{f}_X \\
 \Sigma_R(X) & \xrightarrow{\sigma_R} & R(X)
 \end{array}$$

# Signatures

Let  $\mathbf{R}$  be a monad that models the signature of application and abstraction. Then there exists a unique monad morphism  $\mathbf{f} : \mathbf{L} \rightarrow \mathbf{R}$  which commutes with abstraction and application. Thus, a signature  $\Sigma$  assigns to any monad morphism  $\mathbf{f} : \mathbf{R} \rightarrow \mathbf{R}'$  a family of maps  $(\Sigma(\mathbf{f})_X : \Sigma_R(X) \rightarrow \Sigma_{R'}(X))_X$ .

As for module morphisms, we require that this family commutes with substitution:

$$\Sigma(\mathbf{f})_Y(e[x \mapsto t_x]_{\Sigma_R}) = \Sigma(\mathbf{f})_X(e)[x \mapsto f_X(t_x)]_{\Sigma'_R}$$

Let  $\mathbf{R}$  be a monad that models a signature  $\Sigma$  (there is a module morphism  $\sigma_R : \Sigma_R \rightarrow \mathbf{R}$ ). Then there exists a unique monad morphism  $\mathbf{f} : \mathbf{S} \rightarrow \mathbf{R}$  which commutes with  $\sigma$ :

$$\begin{array}{ccc} \Sigma_L(X) & \xrightarrow{\sigma_L} & L(X) \\ \Sigma(\mathbf{f})_X \downarrow & & \downarrow f_X \\ \Sigma_R(X) & \xrightarrow{\sigma_R} & R(X) \end{array}$$

## PLAN

1. Languages, monads and modules
2. Induction and Initiality
- 3. Signatures**

# Definition of signatures

A **signature**  $\Sigma$  is given by:

- for each monad  $R$ , a module  $\Sigma_R$  over it
- for each monad morphism  $f : R \rightarrow S$ , a family  $\Sigma(f) : \Sigma_R \rightarrow \Sigma_S$  of morphisms which commutes with substitution:

$$\Sigma(f)_Y(e[x \mapsto t_x]_{\Sigma_R}) = \Sigma(f)_X(e)[x \mapsto f_X(t_x)]_{\Sigma'_R}$$

- such that (functoriality)

$$\Sigma(f \circ g) = \Sigma(f) \circ \Sigma(g) \quad \text{and} \quad \Sigma(id_R) = id_{\Sigma_R}$$

A **model** of a signature  $\Sigma$  is a monad  $R$  together with a morphism of modules  $\sigma_R : \Sigma_R \rightarrow R$

A **model morphism** of a signature  $\Sigma$  between two models  $R$  and  $R'$  is a monad morphism  $f : R \rightarrow S$  which commutes with  $\sigma$ :  $\sigma_R \circ f = \Sigma_f \circ \sigma_{R'}$

The **syntax generated by** a signature  $\Sigma$  is its initial model.

# Syntax generated by a signature

This notion of signature is very general so that we do not expect that all of them generate a syntax.

## Examples of syntax generating signatures:

- $R \mapsto R \times R$ :

models are monads  $R$  that comes with a module morphism  $R \times R \rightarrow R$ .

The syntax corresponds to a language with variables and a binary

operator  $b$ :      $\text{expr} ::= x$             *(variable)*  
                        |  $b(t, u)$     *where  $t$  and  $u$  are any expressions*

$$- R \mapsto R \times R + R':$$

By universal property of the disjoint sum  $+$ , models are monads  $R$  equipped with two modules morphisms  $R \times R \rightarrow R$  and  $R' \rightarrow R$ .

## The syntax corresponds to lambda calculus



# Syntax generated by a signature

This notion of signature is very general so that we do not expect that all of them generate a syntax.

## Examples of syntax generating signatures:

- $R \mapsto R \times R$ :

models are monads  $R$  that comes with a module morphism  $R \times R \rightarrow R$ .

The syntax corresponds to a language with variables and a binary

operator  $b$ :      $\text{expr} ::= x$             *(variable)*  
                       |  $b(t, u)$     *where  $t$  and  $u$  are any expressions*

$$- R \mapsto R \times R + R':$$

By universal property of the disjoint sum  $+$ , models are monads  $R$  equipped with two modules morphisms  $R \times R \rightarrow R$  and  $R' \rightarrow R$ .

## The syntax corresponds to lambda calculus