# High-level signatures and initial semantics

#### Ambroise Lafont

joint work with Benedikt Ahrens, André Hirschowitz, Marco Maggesi

### 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  $(t_x)_{x \in X}$  of elements of R(Y), any expression e in R(X) can be substituted to yield an expression  $e[x \mapsto t_x]$  in R(Y).

The substitution is required to statisfy some intuitive equations.

### Operations as module morphisms

In the lambda-calculus,

$$\operatorname{app}(t,u)[x\mapsto v_x] = \operatorname{app}(t[x\mapsto v_x],u[x\mapsto v_x])$$
 i.e.

#### application commutes with substitution

Let us rewrite the right hand side:

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

considering the obvious substitution on pairs of lambda terms.

We abstract this situation as follows:

- pairs of lambda-terms form a module over the lambda-calculus monad,
- 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  $(\mathbf{t_x})_{x \in X}$  of expressions in  $\mathbf{R(Y)}$ , any expression  $\mathbf{e}$  in  $\mathbf{M(X)}$  can be substituted to yield an expression  $\mathbf{e[x} \mapsto \mathbf{t_x}]$  in  $\mathbf{M(Y)}$ .

As for monads, the substitution is required to statisfy 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 R'(X) = R(X + {x}) for any set X of variables
- more generally, we similarly define **M'** given a module **M**

The new variable  $\mathbf{x}$  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:**

 $idM: M \rightarrow M$ 

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

 $app: L \times L \rightarrow L$ 

the application operation of the lambda calculus monad **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  $t \in L(Y + \{x\}) = L'(Y)$ 

**abs:L'** → **L** is a module morphism

### Table of contents

1. Languages, monads and modules

2. Signatures and their models

3. Recursion

4. Presentables 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  $\sigma_R: \Sigma_R \to R$ .

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**:

expr ::= 
$$x$$
 (variable)  
|  $\mathbf{b}(t, u)$  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 x R  $\rightarrow$  R and R'  $\rightarrow$  R. The syntax corresponds to lambda calculus.

# Algebraic signatures

More generally, any signature of the form  $R \mapsto R' \times R'' \times R''' + R \times R'' \times R''' \times R + ...$  (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 specifiy 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  $(\mathbf{f_R}:\Sigma_R\to\Psi_R)_R$  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 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 siupprimer ?)

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

```
expr ::= x (variable)
| t u (application)
| λx.t (abstraction)
```

- the (untyped) syntax of lambda-calculus modulo betaequivalence 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  $var_X(x) = \underline{x} \in A(X)$
- 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:

COMPOSITION OF SUBSTITUTIONS  $e[x \mapsto t_x][y \mapsto u_y] = e[x \mapsto t_x[y \mapsto u_y]]$ 

**IDENTITY SUBSTITUTION** 

$$e[x \mapsto x] = e$$

**VARIABLE SUBSTITUTION** 

$$\forall x \in X \ x[y \mapsto t_y] = t_x$$

# Overview of the methodology

- 1. Introduce a notion of signature.
- 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.

 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:

$$\forall X, \ add_X : A(X) \times A(X) \to A(X)$$

$$(e, e') \mapsto e + e'$$

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"  $\nearrow$  A x A is a **module** on A add commutes with substitution  $\nearrow$  add is a **module morphism** 

### Examples of monads

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

$$\forall X, \ var_X : X \to \mathcal{P}(X)$$
$$x \mapsto \{x\}$$

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:

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

### Induction

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

We compute it by induction on the syntax:

$$fv(x) = \{x\}$$
 (variable)  
 $fv(tu) = fv(t) \cup fv(u)$  (application)  
 $fv(\lambda x.t) = fv(t) \setminus \{x\}$  (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:

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

(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'

$$app_{\mathcal{P}}: \mathcal{P} \times \mathcal{P} \to \mathcal{P}$$

$$(V, V') \mapsto V \cup V'$$

$$abs_{\mathcal{P},X}: \mathcal{P}'(X) \to \mathcal{P}$$

$$V \mapsto V \setminus \{n\}$$

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

$$fv(x) = \{x\}$$
 (commutation with variable)  
 $fv(tu) = fv(t) \cup fv(u)$  (commutation with application)  
 $fv(\lambda x.t) = fv(t) \setminus \{x\}$  (commutation with abstraction)

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



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

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

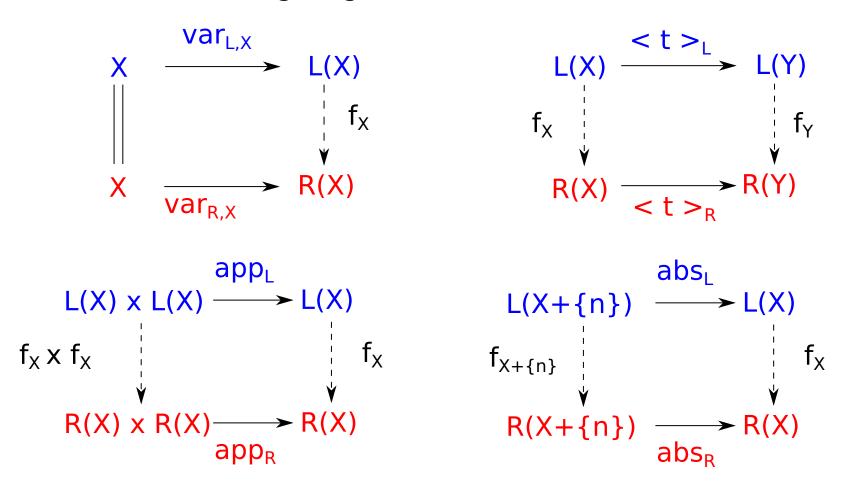
$$R(X) \xrightarrow{< t>_R} R(Y)$$

$$R(X) \times R(X) \longrightarrow R(X)$$
 $app_R$ 

$$R(X+\{n\}) \xrightarrow{abs_R} R(X)$$

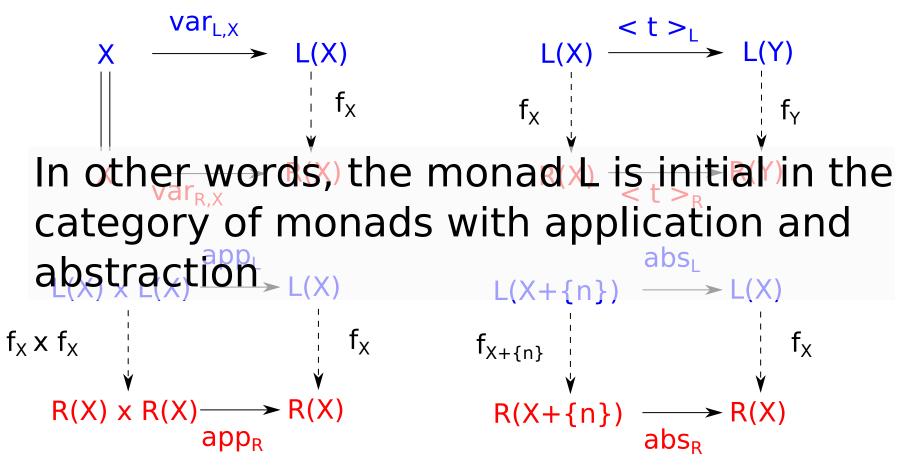
More generally, let R be a monad with application and abstraction.

Then there is a unique family  $(f_X)_X$  of maps (defined by induction) that makes the following diagrams commute:



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:



### 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**.

#### What a signature should be:

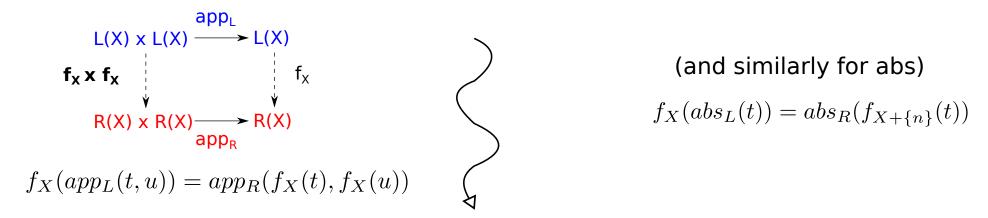
L is initial among the monads R that model the signature  $\Sigma L$  of application and abstraction, i.e. monads R that come with module morphisms:

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:

$$\sigma_R:\Sigma_R\to R$$

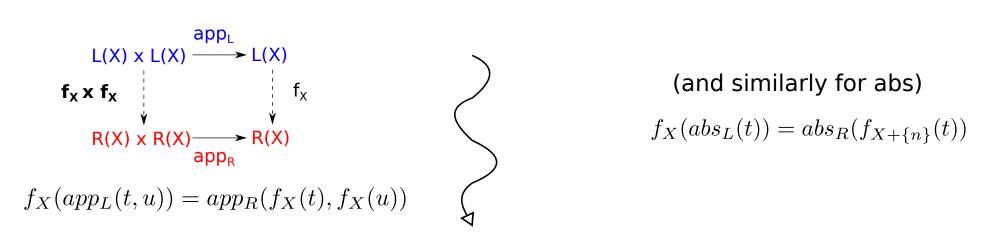
Thus, a signature  $\Sigma$  should assign to any monad R a module  $\Sigma_R$  over it.

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

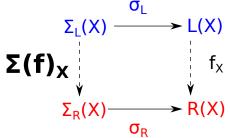


Let  $\mathbf{R}$  be a monad that models a signature  $\Sigma$  (there is a module morphism  $\sigma_R:\Sigma_R\to\mathbf{R}$ ). Then there exists a unique monad morphism  $\mathbf{f}:\mathbf{S}\to\mathbf{R}$  which commutes with  $\sigma$ :

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



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



Thus, a signature  $\Sigma$  assigns to any monad morphism  $f: R \to R'$  a family of maps  $(\Sigma(f)_X : \Sigma_R(X) \to \Sigma_{R'}(X))_{X.}$ 

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

$$\Sigma(f)_Y(e[x\mapsto t_x]_{\Sigma_R})=\Sigma(f)_X(e)[x\mapsto f_X(t_x)]_{\Sigma_R'}$$
 Let **R** be a monad that models a signature **\Sigma** (there is a module morphism

 $\sigma_R: \Sigma_R \to R$ ). Then there exists a unique monad morphism  $f: S \to R$  which

$$\Sigma_{L}(X) \xrightarrow{\sigma_{L}} L(X)$$

$$\Sigma(\mathbf{f})_{X} \downarrow \qquad \qquad \downarrow f_{X}$$

$$\Sigma_{R}(X) \xrightarrow{\sigma_{R}} R(X)$$

### Plan

### **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  $f: R \to S$ , a family  $\Sigma(f): \Sigma_R \to \Sigma_S$  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)

$$\Sigma(f \circ g) = \Sigma(f) \circ \Sigma(g)$$
 and  $\Sigma(id_R) = id_{\Sigma R}$ 

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

A **model morphism** of a signature  $\Sigma$  between two models R and R' is a monad morphism  $f: R \to S$  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 x R  $\rightarrow$  R.

The syntax corresponds to a language with variables and a binary

operator b: 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 x 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 x R  $\rightarrow$  R.

The syntax corresponds to a language with variables and a binary

operator b: 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 x R  $\rightarrow$  R and R'  $\rightarrow$  R.

The syntax corresponds to lambda calculus