High-level signatures and initial semantics

Ambroise Lafont

joint work with Benedikt Ahrens, André Hirschowitz, Marco Maggesi
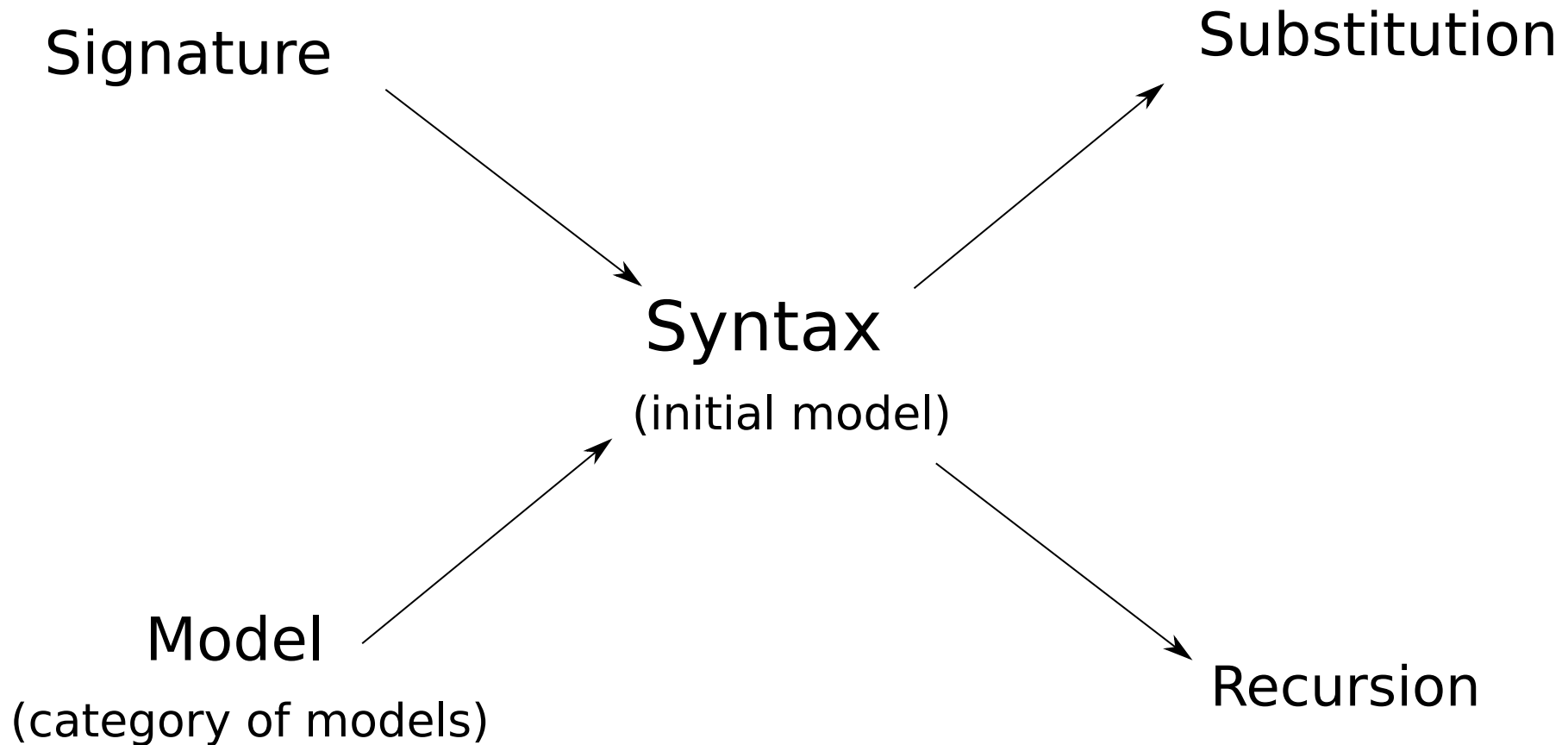
CSL 2018

Introduction

Purpose of our work: specify and construct untyped syntaxes with variables and a well-behaved substitution (e.g. lambda-calculus).

(We expect that our work straightforwardly generalizes to simply-typed syntaxes)

What is a syntax?

Signature

Substitution

Syntax
(initial model)

Model
(category of models)

Recursion

Signatures which we care about: those whose category of models have an *initial object*.

Our work

We present a notion of signature (based on the notion of module over a monad) and their associated models.

Goal of our work: Identify a large class of these signatures whose category of models have an initial object.

Table of contents

1. Standard signatures and their models

2. Languages, monads and modules

3. Recursion

4. Presentables signatures

Example

Consider the syntax generated a binary operation **+** and two constants **0** and **1** (and variables):

$$
\begin{array}{lll}
\text{expr ::= x} & \textit{(variable)} \\
\quad | \ t_1 + t_2 & \textit{(binary operation)} \\
\quad | \ 0 \ | \ 1 & \textit{(constants)}
\end{array}
$$

The syntax induces an endofunctor **B** (on Set) mapping a set of variables to the set of expressions built out of them.

$$B(\emptyset) = \{0, 1, 0 + 1, \dots\}$$

$$B(\{x, y\}) = \{0, 1, 0 + 1, \dots, x, y, x + y, \dots\}$$

Example

The binary operation construction induces a natural transformation:

$$\mathbf{B} \times \mathbf{B} \to \mathbf{B}$$

The constants **0** and **1** induce two natural transformations:

$$\mathbf{1} \to \mathbf{B}$$

Using disjoint union, they gather into a single natural transformation:

$$\mathbf{B} \times \mathbf{B} + \mathbf{1} + \mathbf{1} \to \mathbf{B}$$

Thus, **B** is an algebra for the endofunctor $\mathbf{F} \mapsto \mathbf{F} \times \mathbf{F} + \mathbf{1} + \mathbf{1}$.

Actually, **B** is the initial algebra.

Binding signatures

[Fiore-Plotkin-Turi 1999] ([FPT]) studies binding signatures, given by a family of lists of natural numbers.

Each list specifies an operation in the syntax:
- the size of the list is the number of arguments of the operation
- each natural number in the list indicates the number of bound variables in the corresponding argument

0-aire operations 0 and 1   addition (2 arguments)

Syntax with 0, 1, +: $\big((), (),\qquad\qquad (0, 0)\big)$

Lambda calculus: $\big((0, 0),\qquad\qquad (1)\big)$

application (2 arguments)   lambda-abstraction (1 argument binding 1 variable)

High-level signatures following [FTP]

In the same spirit as the previous example, it is possible to turn any binding signature into an endofunctor $\Sigma_\bullet$.

A model $\mathbf{M}$ is a Σ-algebra equipped with a well-behaved substitution (i.e. the underlying endofunctor is a **monad**).

The action $\Sigma(\mathbf{M}) \to \mathbf{M}$ is required to commute with substitution. This condition requires that the endofunctor $\Sigma(\mathbf{M})$ come with a substitution.

This is the case for endofunctors Σ coming from binding signatures because they have a *strength*.

The initial algebra for a binding signature exists and yields an initial model.

Table of contents

Our signatures

In our framework, signatures are functors, not endofunctors: the source/target category are refined.

Source category:

The domain category of the signature is the subcategory of endofunctors with a well-behaved notion of substitution (i.e. **monads**).

Target category:

A signature Σ associates to any monad $\mathbf{R}$ an endofunctor $\Sigma(\mathbf{R})$ compatible with the monad substitution (i.e. a **module** over the input monad $\mathbf{R}$)

Category of models:

A model is a monad $\mathbf{R}$ with a natural transformation from $\mathbf{\Sigma}(\mathbf{R})$ to $\mathbf{R}$ compatible with the substitution (i.e. a **module morphism**).

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

The substitution is required to statisfy some intuitive equations.

A **monad morphism** between two monads **R** and **S** is a family of maps $(\mathbf{f}_X : \mathbf{R}(\mathbf{X}) \to \mathbf{S}(\mathbf{X}))_X$ preserving variables and substitution.

Operations as module morphisms

In the lambda-calculus,

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

i.e.

application commutes with substitution

Let us rewrite the right hand side:

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

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 **R(Y)**, any expression **e** in **M(X)** can be substituted to yield an expression $\mathbf{e[x \mapsto t_x]}$ in **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 **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 **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

Signatures

A **signature** Σ assigns (functorially) to each monad R a module Σ_R over it.

A **model** of a signature Σ 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 σ).

The **syntax generated by** a signature Σ 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 ↦ R x R

    models are monads R that comes with a module morphism R x R → R.

    The syntax corresponds to a language with variables and a binary

    operator **b**:

$$\text{expr} ::= x \qquad \textit{(variable)}$$
$$| \ \mathbf{b}(t, u) \quad \text{where } t \text{ and } u \text{ are any expressions}$$

- R ↦ R x R + R'

    By universal property of the disjoint sum +, models are monads R

    equipped with two modules morphisms R x R → R and R' → R.

    The syntax corresponds to lambda calculus.

Algebraic signatures

More generally, any signature of the form R ↦ R' x R'' x R''' + R x R'' x R''' x R + ... (i.e. any disjoint sum of products of finite derivatives of the monad) generates a syntax.

These **algebraic signatures** correspond to combinatorial 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

Copie de Recursion

Question: cette section est-elle vraiment éncessaire, puisque on n'a pas vraiment de contribution là-dedans ? (ca fait partie du background initial semantics, non ?).

Benedikt suggère de ne pas faire plus d'une slide là dessus

Recursion

Initiality of the syntax allows recursion.

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

Let **LC** be the monad of lambda-calculus.

Let $t \in$ **LC**(**X**) be a term (whose free variables are in **X**). We want to compute its set of free variables **fv**(t) $\subset$ **X** (i.e. **fv**(t) $\in \mathscr{P}$(**X**)).

Strategy:

The only thing to do is to give the assignement **X** $\mapsto \mathscr{P}$(**X**) the adequate structure of a monad, then of a model.

define **fv** : **LC** $\to \mathscr{P}$ by initiality of LC in the category of models of its signature.

Computing free variables

The assignement which to any set **X** associates its power set $\mathscr{P}(\mathbf{X})$ can be given the structure of a monad (variables are singletons, substitution is union).

$\mathbf{app}_{\mathscr{P}} : \mathscr{P}(\mathbf{X}) \times \mathscr{P}(\mathbf{X}) \to \mathscr{P}(\mathbf{X})$ and $\mathbf{abs}_{\mathscr{P}} : \mathscr{P}(\mathbf{X}+\{\mathbf{x}\}) \to \mathscr{P}(\mathbf{X})$ should be given to yield a model. Let us study the case of **app**:

expected equation of *fv*

$$fv(\mathrm{app}(t, u)) \quad = \quad {\color{blue}fv(t)} \cup {\color{red}fv(u)}$$

$$\mathrm{app}_{\mathcal{P}}({\color{blue}fv(t)}, {\color{red}fv(u)})$$

fv should be a model morphism

Computing free variables

The assignement which to any set **X** associates its power set $\mathscr{P}(\mathbf{X})$ can be given the structure of a monad (variables are singletons, substitution is union).

$\mathbf{app}_{\mathscr{P}} : \mathscr{P}(\mathbf{X}) \times \mathscr{P}(\mathbf{X}) \to \mathscr{P}(\mathbf{X})$ and $\mathbf{abs}_{\mathscr{P}} : \mathscr{P}(\mathbf{X}+\{\mathbf{x}\}) \to \mathscr{P}(\mathbf{X})$ should be given to yield a model. Let us study the case of **app**:

expected equation of *fv*

$$fv(\mathrm{app}(t,u)) \quad = \quad {\color{blue}fv(t)} \cup {\color{red}fv(u)}$$

Thus, we pose:

$$\mathrm{app}_{\mathcal{P}}({\color{blue}fv(t)}, {\color{red}fv(u)})$$

$$\mathrm{app}_{\mathcal{P}}({\color{blue}A}, {\color{red}B}) := {\color{blue}A} \cup {\color{red}B}$$

fv should be a model morphism

Computing free variables

The case of **abs**$_{\mathscr{P}}$ is similar.

It can be shown that **app**$_{\mathscr{P}}$ and **abs**$_{\mathscr{P}}$ are module morphisms, hence give the monad $\mathscr{P}$ the structure of a model for the signature of the lambda-calculus.

By initiality of the syntax **LC**, we get a (unique) model morphism from **LC** to $\mathscr{P}$ which satisfies:

$$fv(t\,u) = fv(t) \cup fv(u)$$
$$fv(\lambda x.t) = fv(t)\backslash\{x\}$$
$$fv(x) = \{x\}$$

Table of contents

Quotient of a signature

Quotient of a set:

A quotient of a set X is a set Y together with a surjection p : X → Y.

$$x \sim x' \qquad \Longleftrightarrow \qquad p(x) = p(x')$$

Quotient of a signature:

A quotient of a signature **Σ** is a signature **Ψ** together with a (natural) family of module morphisms $(\mathbf{f_R} : \mathbf{\Sigma_R} \to \mathbf{\Psi_R})_\mathbf{R}$ 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 a syntax generated by an algebraic (or combinatorial) signature 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.

A syntactic closure operator:

Such an operator allows to bind a given set of variables in an expression (thus invariant under permutation of these variables).

The signature is obtained as a quotient of the algebraic signature specifying a sequence of increasingly sequential binding operators.

Examples of presentable signatures

Explicit substitution:

It is possible to specify an operation $\_\langle \mathbf{x_i} \mapsto \mathbf{t_i} \rangle$ that mimics the behavior of the true substitution $\_[\mathbf{x_i} \mapsto \mathbf{t_i}]$ in the sense that it enjoys some of its coherences, for example:

- if **u** does not depend on **y**,

$$u\langle x \mapsto v, y \mapsto w \rangle = u\langle x \mapsto v \rangle$$

- let **u'** be **u** where the variables **x** and **y** have been swapped,

$$u'\langle x \mapsto v, y \mapsto w \rangle = u\langle x \mapsto w, y \mapsto v \rangle$$

Examples of presentable signatures

A coherent fixedpoint operator:

A language with (mutual) fixedpoints comes with a construction

```
let rec f₁ = t₁
    and f₂ = t₂            where each fⱼ may appear as a
          ...              variable in each expression tᵢ.
    and fₙ = tₙ
in fᵢ
```

where each $\mathbf{f}_j$ may appear as a variable in each expression $\mathbf{t}_i$.

Thus, it takes **n** expressions $\mathbf{t}_1,..,\mathbf{t}_n$ depending on **n** new variables $\mathbf{f}_1,..,\mathbf{f}_n$ and produces an expression which does not depend on these variables.

As such, it can be specified by an algebraic signature.

Coherent fixedpoint operator

But we would like to encode some of the expected behaviour of such a fixed point. For instance:

$$
\begin{array}{l}
\texttt{let rec } \mathbf{f}_1 = \mathbf{t}_1 \\
\quad \texttt{and } \mathbf{f}_2 = \mathbf{t}_2 \\
\texttt{in } \mathbf{f}_1
\end{array}
\;=\;
\begin{array}{l}
\texttt{let rec } \mathbf{f}_1 = \mathbf{t}_2' \\
\quad \texttt{and } \mathbf{f}_2 = \mathbf{t}_1' \\
\texttt{in } \mathbf{f}_1
\end{array}
$$

($\mathbf{t}_i{}'$ is $\mathbf{t}_i$ where $\mathbf{f}_1$ and $\mathbf{f}_2$ have been swapped)

or, if $\mathbf{t}_1$ does not depend on $\mathbf{f}_2$,

$$
\begin{array}{l}
\texttt{let rec } \mathbf{f}_1 = \mathbf{t}_1 \\
\quad \texttt{and } \mathbf{f}_2 = \mathbf{t}_2 \\
\texttt{in } \mathbf{f}_1
\end{array}
\;=\;
\begin{array}{l}
\texttt{let rec } \mathbf{f}_1 = \mathbf{t}_1 \\
\texttt{in } \mathbf{f}_1
\end{array}
$$

A construction satisfying these equations can be specified by quotienting the naive algebraic signature.

Conclusion

We found a criterion for high-level 'monadic' signatures to specify a syntax. This criterion encompasses the classical combinatorial signatures, and allows fancier operations at the level of the syntax.

Future work:

We plan to take into account more sophisticated equations in the syntax than just quotients, extend our framework to simply typed syntaxes.

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).

First-order signatures

Any first-order signature induces similarly an endofunctor (on the category of endofunctors on Set).

Natural notion of model for an endofunctor: an algebra for this endofunctor.

The syntax is characterized as the initial algebra/model.

Example : binary trees (leaves labelled by variables)

The language induces an endofunctor $\mathbf{B}$ (on Set) mapping a set of variables to the set of binary trees built out of them.

The binary/unary node constructions induce natural transformations:

$$\mathbf{B} \times \mathbf{B} \rightarrow \mathbf{B} \qquad\qquad \mathbf{B} \rightarrow \mathbf{B}$$

binary node (2 arguments) $\qquad\qquad$ unary node (1 argument)

Using disjoint union, they gather into a single one:

$$\mathbf{B} \times \mathbf{B} + \mathbf{B} \rightarrow \mathbf{B}$$

Thus, $\mathbf{B}$ is an algebra for the endofunctor $\mathbf{F} \mapsto \mathbf{F} \times \mathbf{F} + \mathbf{F}$. Actually, $\mathbf{B}$ is the initial algebra.

Signatures with strength

In the litterature, there is also a refined notion of signature as an endofunctor with *strength*.

This extra property allow in particular to give a refined notion of model as algebras with a well-behaved substitution.

A combinatorial signature yields such a signature.

We don't need the notion of strength in our framework.

First-order signatures

A first-order signature is a family of natural numbers.

Each natural number specifies the arity (number of arguments) of the associated operation.

Example : binary trees (leaves are labelled by variables)

expr ::= x   *(variable)*
      | $b(t_1,t_2)$ *(node with 2 children)*
      | $u(t)$ *(node with 1 child)*

Two operations:
  binary node (2 arguments)   unary node (1 argument)

$$(2 , 1)$$

Copie de High-level signatures

As before, binding signatures induce an endofunctor.

Category of models of such a signature: the category of its algebras.

Any combinatorial signature can be turned into an endofunctor on the category of endofunctors on **Set**.

Indeed, we think of an endofunctor on **Set** as a mapping associating to any set of free variables the set of expressions built out of them.

Copie de 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.

Ultimate goal: (untyped) languages with variables and substitution

Classical signatures: a combinatorial specification of a language

as families 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:

application (2 arguments)

lambda-abstraction (1 argument binding 1 variable)

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

This work: a general notion of signature

A coherent fixedpoint operator:

A unary fixed point operator for a monad is an operation (or module morphism) **Y** turning an expression **e** depending on an additional formal variable ★ into an expression **Y(e)** that doesn't depend on it, such that the fixedpoint equation holds:

$$Y(e[\star \mapsto Y(e)]) = Y(e)$$

(for the lambda-calculus monad modulo beta-eta equivalence, it corresponds to a fixedpoint combinator)

Examples of monads (à siupprimer ?)

- the syntax of arithmetic expressions

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

$$
\begin{aligned}
\text{expr} ::= \ &x &&\textit{(variable)}\\
| \ &t \ u &&\textit{(application)}\\
| \ &\lambda x.t &&\textit{(abstraction)}
\end{aligned}
$$

- 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.

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 $\mathrm{var}_X(x) = \underline{x} \in A(X)$

- 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 $\quad e[x \mapsto t_x][y \mapsto u_y] = e[x \mapsto t_x[y \mapsto u_y]]$

IDENTITY SUBSTITUTION $\qquad\qquad\qquad\qquad e[x \mapsto x] = e$

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

Copie de 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**

:

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

$$\forall X,\ add_X : A(X) \times A(X) \rightarrow A(X)$$
$$(e, e') \mapsto e + e'$$

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) x A(X) expressions are "*substitutable*" ⟿ A x A is a **module** on A

add commutes with substitution ⟿ *add* is a **module morphism**

Examples of monads

- the assignement X $\mapsto$ $\mathscr{P}$(X) = { U | U $\subset$ X } yields a monad $\mathscr{P}$:

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

Let U $\subset$ X (i.e. U $\in$ $\mathscr{P}$(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:

$$fv(x) = \{x\} \qquad\qquad\qquad\qquad \text{(variable)}$$
$$fv(tu) = fv(t) \cup fv(u) \qquad\qquad \text{(application)}$$
$$fv(\lambda x.t) = fv(t) \backslash \{x\} \qquad\qquad \text{(abstraction)}$$

This is formalized in our setting as a family of maps *(fv$_X$: L(X) → $\mathscr{P}$(X))$_X$*
which *commutes with variable and substitution:*

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

(This is a definition of a monad morphism)

Example: computing the free variables of a lambda-term

fv also commutes with 'application' and 'abstraction'

$$app_{\mathcal{P}} : \mathcal{P} \times \mathcal{P} \to \mathcal{P}$$
$$(V, V') \mapsto V \cup V'$$

$$abs_{\mathcal{P}, X} : \overbrace{\mathcal{P}'(X)}^{\mathcal{P}(X + \{n\})} \to \mathcal{P}$$
$$V \mapsto V \backslash \{n\}$$

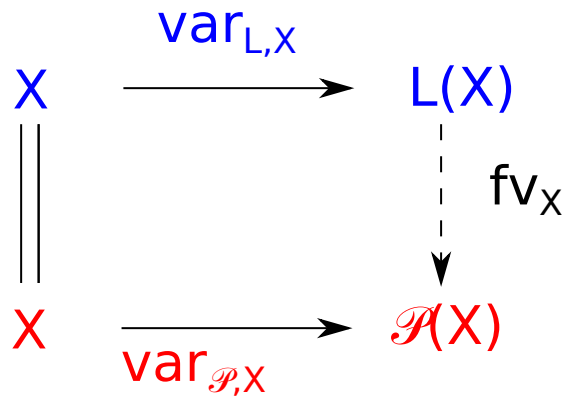Actually, these commutations **define** *fv* uniquely by induction:

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

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

More generally, let R be a monad with application and abstraction.
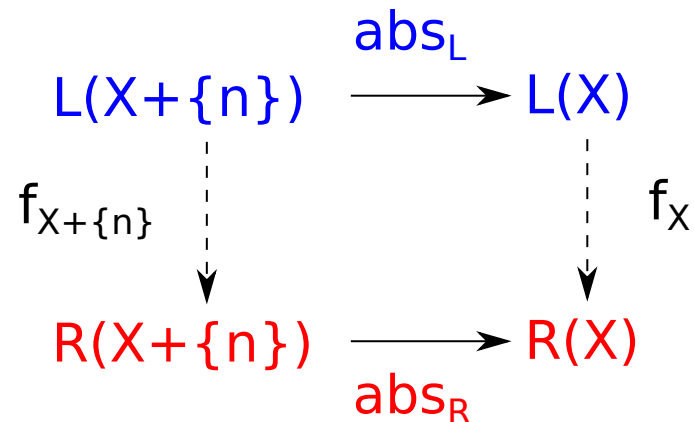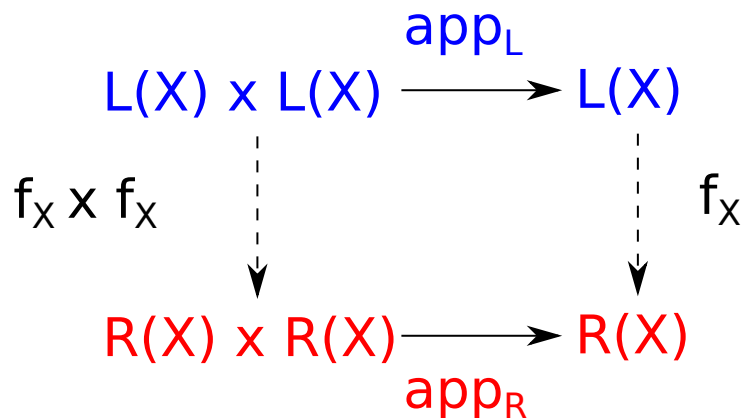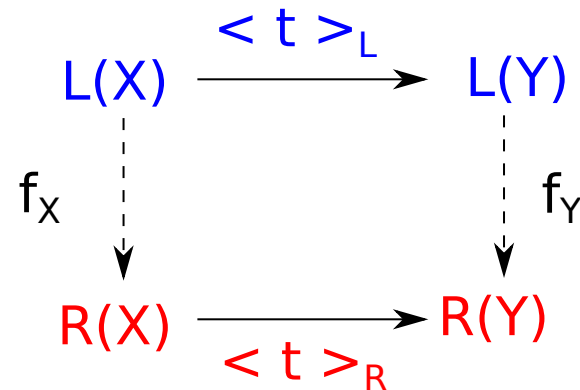
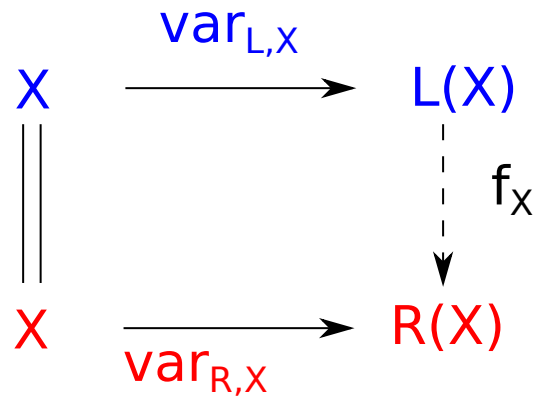$$X \xrightarrow{\;\text{var}_{R,X}\;} R(X)$$

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

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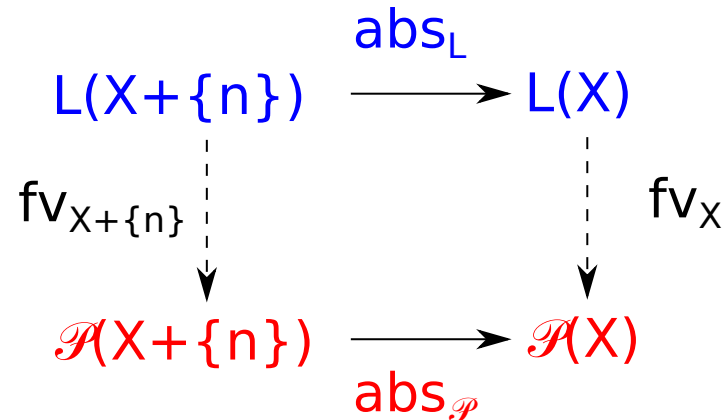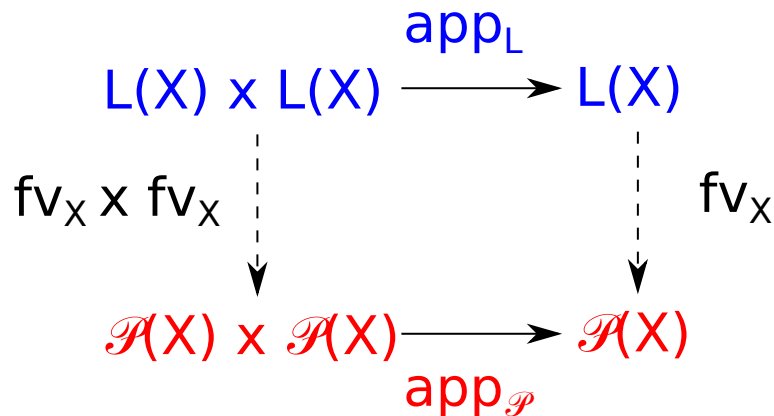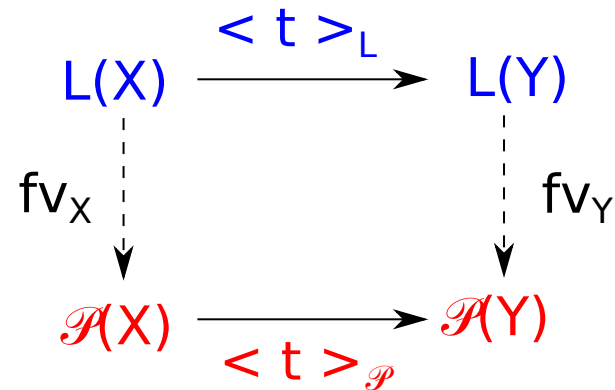
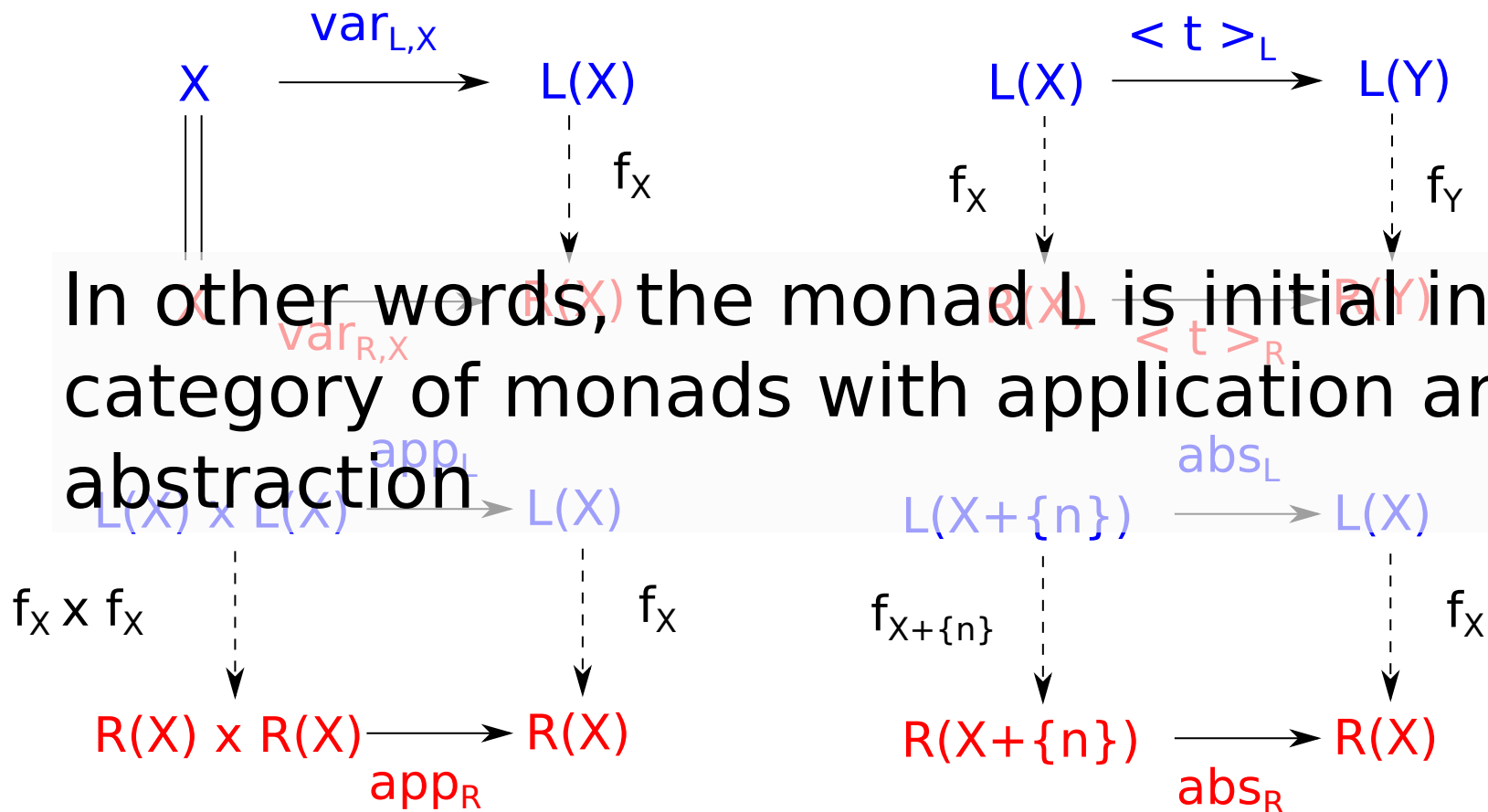$$R(X+\{n\}) \xrightarrow{\;\text{abs}_R\;} R(X)$$

Induction and initiality

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

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

$$X \xrightarrow{\ \mathsf{var}_{L,X}\ } L(X)$$

$$L(X) \xrightarrow{\ <t>_L\ } L(Y)$$

f_X f_X f_Y

$\mathsf{var}_{R,X}$ $<t>_R$

app_L abs_L

$$L(X) \times L(X) \longrightarrow L(X)$$

$$L(X+\{n\}) \longrightarrow L(X)$$

$f_X \times f_X$ f_X $f_{X+\{n\}}$ f_X

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

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

In other words, the monad L is initial in the
category of monads with application and
abstraction

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 ΣL of application and abstraction, i.e. monads R that come with module morphisms:

$$app_R : R \times R \to R$$
$$abs_R : R' \to R$$

or

$$[app_R, abs_R] : \underbrace{R \times R + R'}_{\Sigma L(R)} \to R$$

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

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

Thus, a signature Σ should assign to any monad R a module Σ_R over it.

Signatures

Let **R** be a monad that models the signature of application and abstraction. Then there exists a unique monad morphism **f** : **L** → **R** which commutes with abstraction and application:

L(X) x L(X) $\xrightarrow{\text{app}_L}$ L(X)

$f_X \times f_X$ ↓ ↓ f_X

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

$$f_X(app_L(t, u)) = app_R(f_X(t), f_X(u))$$

(and similarly for abs)

$$f_X(abs_L(t)) = abs_R(f_{X+\{n\}}(t))$$

Let **R** be a monad that models a signature **Σ** (there is a module morphism **σ**R : **Σ_R** → **R**). Then there exists a unique monad morphism **f** : **S** → **R** which commutes with **σ**:

$\Sigma_L(X) \xrightarrow{\sigma_L} L(X)$

?? ↓ ↓ f_X

$\Sigma_R(X) \xrightarrow{\sigma_R} R(X)$

Signatures

Let **R** be a monad that models the signature of application and abstraction. Then there exists a unique monad morphism **f** : **L** → **R** which commutes with abstraction and application:

$$L(X) \times L(X) \xrightarrow{\quad app_L \quad} L(X)$$

$f_X \times f_X$ $\qquad f_X$

$$R(X) \times R(X) \xrightarrow{\quad app_R \quad} R(X)$$

$$f_X(app_L(t,u)) = app_R(f_X(t), f_X(u))$$

(and similarly for abs)

$$f_X(abs_L(t)) = abs_R(f_{X+\{n\}}(t))$$

Let **R** be a monad that models a signature **Σ** (there is a module morphism **σ**R : **Σ_R** → **R**). Then there exists a unique monad morphism **f** : **S** → **R** which commutes with **σ**:

$$\Sigma_L(X) \xrightarrow{\quad \sigma_L \quad} L(X)$$

$\mathbf{\Sigma(f)_X}$ $\qquad f_X$

$$\Sigma_R(X) \xrightarrow{\quad \sigma_R \quad} R(X)$$

Signatures

Thus, a signature Σ assigns to any monad morphism f : R → 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:

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

PLAN

1. Languages, monads and modules

2. Induction and Initiality

3. Signatures

Definition of signatures

A **signature** Σ is given by:

- for each monad R, a module Σ_R over it

- for each monad morphism f : R → S, a family Σ(f) : Σ_R → Σ_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)

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

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

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

The **syntax generated by** a signature Σ 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: 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 $R \times R \to R$.

  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 $R \times R \to R$ and $R' \to R$.

  The syntax corresponds to lambda calculus