High-level signatures and initial semantics

Ambroise Lafont

joint work with Benedikt Ahrens, André Hirschowitz, Marco Maggesi

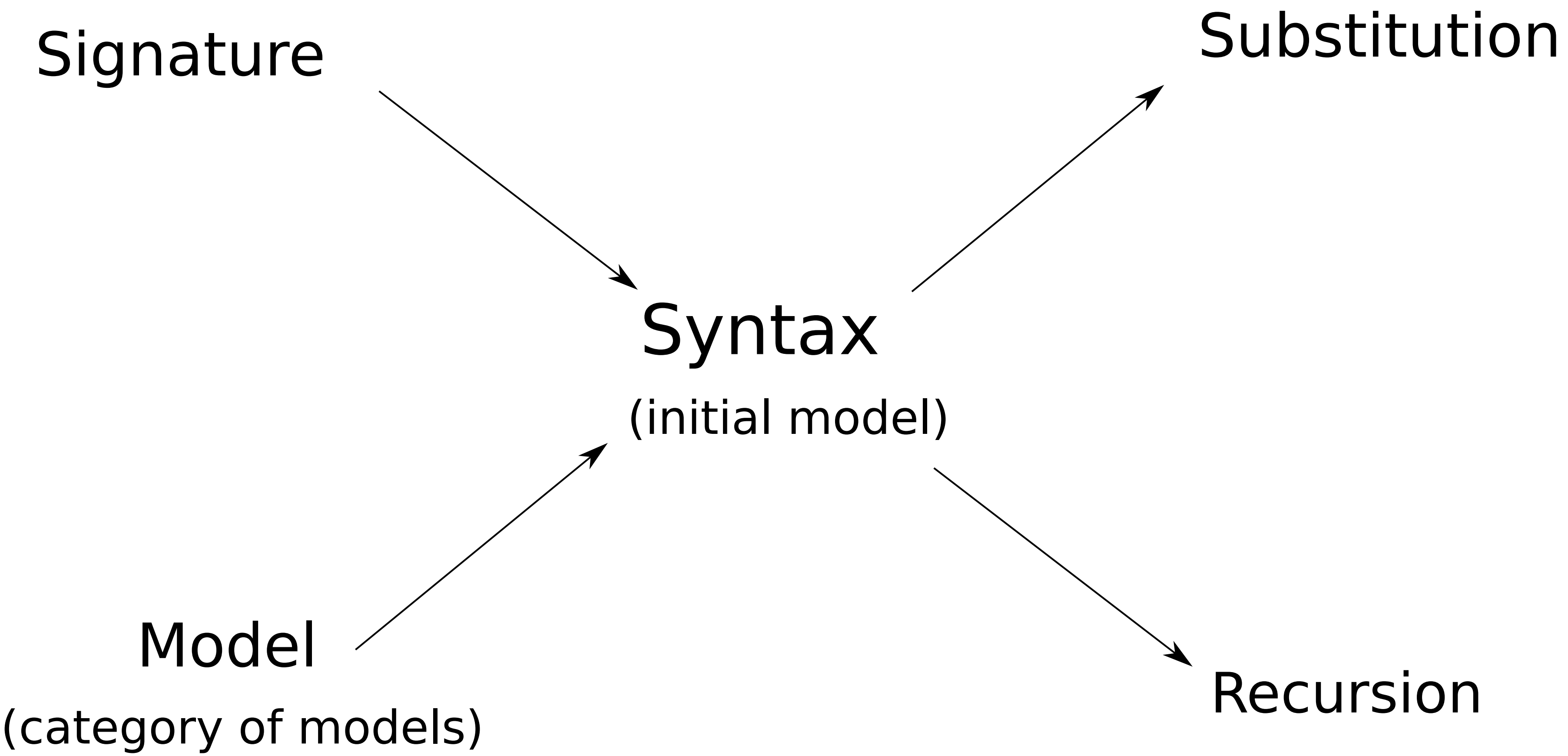
CSL 2018

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?



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

Our work

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

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

Table of contents

Example: 0, ★

Consider the syntax generated by a binary operation ★ and a constant **0** (and variables):

expr ::=	x	*(variable)*
	| t_1 ★ t_2	*(binary operation)*
	| 0	*(constant)*

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

$$B(\emptyset) = \{0, 0 \star 0, \ldots\}$$

$$B(\{x, y\}) = \{0, 0 \star 0, \ldots, x, y, x \star y, \ldots\}$$

Example: 0, ★

The binary operation construction induces a natural transformation:

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

The constant **0** induces a natural transformation:

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

Variables induce a natural transformation

$$\mathbf{Id}_{\mathrm{Set}} \to \mathbf{B}$$

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

$$\mathbf{B} \times \mathbf{B} + \mathbf{1} + \mathbf{Id}_{\mathrm{Set}} \to \mathbf{B}$$

i.e. **B** is an algebra for the endofunctor $\mathbf{F} \mapsto \mathbf{F} \times \mathbf{F} + \mathbf{1} + \mathbf{Id}_{\mathrm{Set}}$ on the category $\mathbf{End}_{\mathrm{Set}}$ of endofunctors on Set**.**

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

[Fiore-Plotkin-Turi 1999]

Binding signature = a family of lists of natural numbers.

Each list specifies an operation in the syntax:

- length of the list = number of arguments of the operation
- natural number in the list = number of bound variables in the corresponding argument

Syntax with 0, $\star$: $\big((), (0, 0)\big)$

0-aire operation **0** binary operation $\star$

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

application lambda-abstraction

Models and signatures following [FPT]

In the same spirit as in the first example $(0, \star)$, any binding signature can be turned into an endofunctor $\mathbf{\Sigma}$ on the category $\mathbf{End}_{\mathrm{Set}}$.

A natural notion of model: $\mathbf{\Sigma} + \mathbf{Id}_{\mathrm{Set}}$ -algebra

Theorem [FPT]: The initial $\mathbf{\Sigma} + \mathbf{Id}_{\mathrm{Set}}$ -algebra of a binding signature $\mathbf{\Sigma}$ exists and comes with a *well-behaved substitution*.

Any endofunctor on $\mathrm{End}_{\mathrm{Set}}$ can be viewed as a signature associated with this notion of model.

Models and signatures following [FTP]

An endofunctor Σ induced by binding signatures comes with a *strength.* It allows to define **Σ-monoids:** $\Sigma + \mathrm{Id}_{\mathrm{Set}}$-algebras equipped with a well-behaved substitution.

Σ-monoid morphisms = algebra morphisms commuting with substitution.

Theorem [FPT]: Initial $\boldsymbol{\Sigma} + \mathbf{Id}_{\mathrm{Set}}$-algebra morphisms commute with substitution (when the target algebra is a Σ-monoid).

In other words, the initial $\boldsymbol{\Sigma} + \mathbf{Id}_{\mathrm{Set}}$-algebra is also initial in this category of models.

Any endofunctor on $\mathrm{End}_{\mathrm{Set}}$ *with strength* can be viewed as a signature (as in [Matthes-Uustalu 2004]) associated with this notion of model.

Table of contents

Our signatures

A signature **Σ** assigns functorially to any endofunctor **R** with substitution (i.e. a **monad**) an endofunctor **Σ**(**R**) compatible with the monad substitution (i.e. a **module** over the input monad).

A model is a monad **R** with a natural transformation from **Σ**(**R**) to **R** compatible with the substitution (i.e. a **module morphism**).

Binding signatures yield signatures. Their category of models are (essentially) equivalent to the [FTP] ones.

Monads

A monad $\mathbf{R}$ corresponds to a language with variables as placeholders for any expression of $\mathbf{R}$.

$\mathbf{R}(\mathbf{X})$ denotes the set of expressions taking variables in $\mathbf{X}$.

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

The substitution is required to satisfy some intuitive equations.

A **monad morphism** between two monads $\mathbf{R}$ and $\mathbf{S}$ 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,

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

application commutes with substitution?

Yes: rewrite the right hand side as:

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

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**.

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

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 modules **M** and **N**

- **M'** is the module defined by **M'**(**X**) = **M**(**X** + {**x**}) for any set **X** of variables 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 $(\mathbf{f}_X:\mathbf{M}(\mathbf{X}) \to \mathbf{N}(\mathbf{X}))_X$ commuting with substitution.

Examples:

$id_M : M \to M$

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

$app : L \times L \to L$

the application operation of the lambda calculus monad **L**.

$abs : L' \to L$

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

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

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 $\sigma : \Sigma_R \to R$.

Models form a category (morphisms are monad morphisms compatible with σ).

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

Notion of signature too general: existence of initial object ?

Examples of syntax generating signatures

- $R \mapsto R \times R + 1$

  By universal property of the disjoint sum, models are monads R equipped with module morphisms $R \times R \to R$ and $1 \to R$. The syntax corresponds to our example with **0** and ★.

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

  Models are monads R equipped with two modules morphisms $R \times R \to R$ and $R' \to R$. The syntax corresponds to lambda calculus.

Algebraic signatures

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

These signatures correspond to binding signatures.

Our main result: quotients of binding signatures also generate a syntax

Table of contents

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

Strategy:

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

define $\mathbf{fv} : \mathbf{LC} \to \mathscr{P}$ 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 $\mathscr{P}$(**X**) can be given the structure of a monad (variables are singletons, substitution is union).

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

expected equation of *fv*

$$fv(\mathrm{app}(t,u)) \;=\; fv(t) \cup fv(u)$$

$$fv(\mathrm{app}(t,u)) \;=\; \mathrm{app}_{\mathcal{P}}(fv(t), fv(u))$$

fv should be a model morphism

Computing free variables

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

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

expected equation of *fv*

$$fv(\mathrm{app}(t,u)) = fv(t) \cup fv(u)$$

$$\| $$

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

fv should be a model morphism

Thus, we pose:

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

Computing free variables

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

It can be shown that $\mathbf{app}_{\mathscr{P}}$ and $\mathbf{abs}_{\mathscr{P}}$ 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:

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

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 $p : X \to Y$.

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

Quotient of a signature:

A quotient of a signature $\mathbf{\Sigma}$ is a signature $\mathbf{\Psi}$ 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 a binding 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 x 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
        ...
    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:

```
let rec f1 = t1              let rec f1 = t2'
    and f2 = t2       =          and f2 = t1'
in f1                        in f1
```

($\mathbf{t_i}$' is $\mathbf{t_i}$ where $\mathbf{f}_1$ and $\mathbf{f}_2$ have been swapped)

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

```
let rec f1 = t1
    and f2 = t2       =      let rec f1 = t1
in f1                        in f1
```

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 binding signatures, and allows fancier operations at the level of the syntax.

Our main theorem have been formalized using the Coq library UniMath.

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).

Copie de Models and signatures following [FTP]

For endofunctors induced by binding signatures, they define the category of algebras equipped with a well-behaved substitution.

The syntax (as endofunctor) is still the same in this category of models: the initial morphism from the syntax to such a model commutes with substitution.

This notion of model works with any endofunctor with a strength, seen as a general notion of signature.

By definition, the initial model (if it exists) comes with a well-behaved substitution.

Copie de Models following [FPT]

In the same spirit as in the first example (0,1,+), any binding signature can be turned into an endofunctor $\mathbf{\Sigma}$ on the category $\mathbf{End}_{\mathrm{Set}}$.

A natural notion of model: $\mathbf{\Sigma} + \mathbf{Id}_{\mathrm{Set}}$-algebra

Theorem [FPT]: The initial model of a binding signature exists and comes with a *well-behaved substitution*.

pubelle

Copie de Examples of presentable signatures

Copie de Introduction

Copie de Purpose of our work

Copie de High-level signatures

Signatures with strength

Copie de First-order signatures

Copie de Example

First-order signatures

Signatures suggested by [FTP]

First-order signatures

Models following [FTP]

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

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

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

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**

Copie de Operations as module morphisms

:

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

$$\begin{aligned} \forall X,\ add_X : A(X) \times A(X) &\to A(X) \\ (e, e') &\mapsto e + e' \end{aligned}$$

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) 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 $X \mapsto \mathscr{P}(X) = \{ U \mid U \subset X \}$ yields a monad $\mathscr{P}$:

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

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:

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

This is formalized in our setting as a family of maps *(*$\mathrm{fv}_X : L(X) \to \mathscr{P}(X))_X$ 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'

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

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

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

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

Induction and initiality

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

$$\begin{tikzcd} X \arrow[r, "\mathrm{var}_{L,X}"] \arrow[d, equal] & L(X) \arrow[d, dashed, "\mathrm{fv}_X"] \\ X \arrow[r, "\mathrm{var}_{\mathscr{P},X}"'] & \mathscr{P}(X) \end{tikzcd}
\qquad
\begin{tikzcd} L(X) \arrow[r, "< t >_L"] \arrow[d, dashed, "\mathrm{fv}_X"'] & L(Y) \arrow[d, dashed, "\mathrm{fv}_Y"] \\ \mathscr{P}(X) \arrow[r, "< t >_{\mathscr{P}}"'] & \mathscr{P}(Y) \end{tikzcd}$$

$$\begin{tikzcd} L(X) \times L(X) \arrow[r, "\mathrm{app}_L"] \arrow[d, dashed, "\mathrm{fv}_X \times \mathrm{fv}_X"'] & L(X) \arrow[d, dashed, "\mathrm{fv}_X"] \\ \mathscr{P}(X) \times \mathscr{P}(X) \arrow[r, "\mathrm{app}_{\mathscr{P}}"'] & \mathscr{P}(X) \end{tikzcd}
\qquad
\begin{tikzcd} L(X+\{n\}) \arrow[r, "\mathrm{abs}_L"] \arrow[d, dashed, "\mathrm{fv}_{X+\{n\}}"'] & L(X) \arrow[d, dashed, "\mathrm{fv}_X"] \\ \mathscr{P}(X+\{n\}) \arrow[r, "\mathrm{abs}_{\mathscr{P}}"'] & \mathscr{P}(X) \end{tikzcd}$$

Induction and initiality

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

$$X \xrightarrow[\mathrm{var}_{R,X}]{} R(X) \qquad R(X) \xrightarrow[<t>_R]{} R(Y)$$

$$R(X) \times R(X) \xrightarrow[\mathrm{app}_R]{} R(X) \qquad R(X+\{n\}) \xrightarrow[\mathrm{abs}_R]{} R(X)$$

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:

$$\begin{tikzcd} X \arrow[r, "var_{L,X}"] \arrow[d, equal] & L(X) \arrow[d, dashed, "f_X"] \\ X \arrow[r, "var_{R,X}"'] & R(X) \end{tikzcd} \qquad \begin{tikzcd} L(X) \arrow[r, "< t >_L"] \arrow[d, dashed, "f_X"'] & L(Y) \arrow[d, dashed, "f_Y"] \\ R(X) \arrow[r, "< t >_R"'] & R(Y) \end{tikzcd}$$

$$\begin{tikzcd} L(X) \times L(X) \arrow[r, "app_L"] \arrow[d, dashed, "f_X \times f_X"'] & L(X) \arrow[d, dashed, "f_X"] \\ R(X) \times R(X) \arrow[r, "app_R"'] & R(X) \end{tikzcd} \qquad \begin{tikzcd} L(X+\{n\}) \arrow[r, "abs_L"] \arrow[d, dashed, "f_{X+\{n\}}"'] & L(X) \arrow[d, dashed, "f_X"] \\ R(X+\{n\}) \arrow[r, "abs_R"'] & R(X) \end{tikzcd}$$

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:

$$
\begin{tikzcd}
X \arrow[r, "var_{L,X}"] \arrow[d, equal] & L(X) \arrow[d, dashed, "f_X"] \\
X \arrow[r, "var_{R,X}"'] & R(X)
\end{tikzcd}
\qquad
\begin{tikzcd}
L(X) \arrow[r, "< t >_L"] \arrow[d, dashed, "f_X"'] & L(Y) \arrow[d, dashed, "f_Y"] \\
R(X) \arrow[r, "< t >_R"'] & R(Y)
\end{tikzcd}
$$

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

$$
\begin{tikzcd}
L(X) \times L(X) \arrow[r, "app_L"] \arrow[d, dashed, "f_X \times f_X"'] & L(X) \arrow[d, dashed, "f_X"] \\
R(X) \times R(X) \arrow[r, "app_R"'] & R(X)
\end{tikzcd}
\qquad
\begin{tikzcd}
L(X+\{n\}) \arrow[r, "abs_L"] \arrow[d, dashed, "f_{X+\{n\}}"'] & L(X) \arrow[d, dashed, "f_X"] \\
R(X+\{n\}) \arrow[r, "abs_R"'] & R(X)
\end{tikzcd}
$$

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:

$$
\begin{tikzcd}
L(X) \times L(X) \arrow[r, "app_L"] \arrow[d, dashed, "\mathbf{f_X \times f_X}"'] & L(X) \arrow[d, dashed, "f_X"] \\
R(X) \times R(X) \arrow[r, "app_R"'] & R(X)
\end{tikzcd}
$$

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

(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**:

$$
\begin{tikzcd}
\Sigma_L(X) \arrow[r, "\sigma_L"] \arrow[d, dashed, "??"'] & L(X) \arrow[d, dashed, "f_X"] \\
\Sigma_R(X) \arrow[r, "\sigma_R"'] & R(X)
\end{tikzcd}
$$

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:

$$\begin{tikzcd} L(X) \times L(X) \arrow[r, "app_L"] \arrow[d, dashed, "\mathbf{f_X \times f_X}"'] & L(X) \arrow[d, dashed, "f_X"] \\ R(X) \times R(X) \arrow[r, "app_R"'] & R(X) \end{tikzcd}$$

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

(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 $\boldsymbol{\sigma}_R$: $\mathbf{\Sigma_R}$ → **R**). Then there exists a unique monad morphism **f** : **S** → **R** which commutes with **σ**:

$$\begin{tikzcd} \Sigma_L(X) \arrow[r, "\sigma_L"] \arrow[d, dashed, "\mathbf{\Sigma(f)_X}"'] & L(X) \arrow[d, dashed, "f_X"] \\ \Sigma_R(X) \arrow[r, "\sigma_R"'] & R(X) \end{tikzcd}$$

Signatures

Thus, a signature Σ 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:

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

$$\begin{tikzcd}
\Sigma_L(X) \arrow[r, "\sigma_L"] \arrow[d, dashed, "\mathbf{\Sigma(f)_X}"'] & L(X) \arrow[d, dashed, "f_X"] \\
\Sigma_R(X) \arrow[r, "\sigma_R"'] & R(X)
\end{tikzcd}$$

PLAN

Definition of signatures

A **signature** Σ is given by:

- for each monad R, a module Σ_R over it
- for each monad morphism $f : R \to S$, 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 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 $f : R \to S$ which commutes with σ: $\sigma_R \circ f = \Sigma_f \circ \sigma_{R'}$

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 $R \times R \to R$.

  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

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