High-level signatures and initial semantics

B. Ahrens, A. Hirshowitz, A. Lafont, M. Maggesi

Language with substitutions

Goal of our work: construct the *syntax* associated to a large class of *signatures*.

Example of a "language" with variables and substitution (i.e. replacing variables with any expression yields a valid expression): formal arithmetic expressions with +, ×, natural numbers.

$$x + (y \times 3) \quad \xmapsto[x \mapsto 2,\, y \mapsto z+5]{\text{substitution}} \quad 2 + ((z+5) \times 3)$$

Other example: lambda-calculus

Abstract

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 should 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**

PLAN

1. **Languages, monads and modules**

2. Signatures and their models

3. Recursion

4. Presentables signatures

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

Examples of monads

- the syntax of arithmetic expressions

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

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

L(∅) = {closed terms} = { λx.x, λx.λy.(x y), (λx.x x)(λx.x x), ...}
L({z}) = {z, λx.z, λx.(x z), ..} ∪ L(∅)

$$(\lambda x.xz)[z \mapsto \lambda y.y] = \lambda x.x(\lambda y.y)$$

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

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:

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

Operations as module morphisms

Arithmetic operations as module morphisms:

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

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

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) x A(X) expressions are "*substitutable*" $\rightsquigarrow$ A x A is a ***module*** on A

add commutes with substitution $\rightsquigarrow$ *add* is a ***module morphism***

Module over a monad

Module over a monad:

A module **M** over the monad **A** corresponds to expressions where variables can be substituted with expressions of the language A.

- it associates a set M(X) to any set X: M(X) can be thought of as "generalized" expressions taking variables in X.

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

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. the assignement X $\mapsto$ R(X) x R(X))

- **M x N** for any module M and N

Important example : Derivative of a module

- **X $\mapsto$ R(X + {n})** where $n \notin X$ yields a module denoted by **R'**
- more generally, we similarly define **M'** given a module **M**

The new variable n $\notin$ X is helpful for modelling 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 $(f_X : M(X) \to N(X))_X$ that commutes with substitution:

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

- **$\mathbf{app : L \times L \to L}$** denoting the application operation of the lambda calculus monad L: $app(t,u) = t\ u$

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

PLAN

1. Languages, monads and modules

2. Signatures and their models

3. Recursion

4. Presentables signatures

Signatures

A **signature $\boldsymbol{\Sigma}$** assigns (functorially) to each monad **R** a module $\boldsymbol{\Sigma_R}$ 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 signatures

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

Algebraic signatures

More generally, any signature of the form $R \mapsto R' \times R'' \times R''' + R \times R'' \times R''' \times R + \ldots$ (i.e. any disjoint sum of products of finite derivatives of the monad) generates a syntax. We call them **algebraic signatures**: they correspond to languages with n-ary operations that can bind a finite number of variables in their arguments.

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

Example:

- $R \mapsto (R \times R) / S_2$ associates to any monad R the module of its unordered pairs. Models (in particular the syntax) are monads equipped with a binary *commutative* operation.

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. We call them **algebraic signatures**: they correspond to languages with n-ary operations that can bind a finite number of variables in their arguments.

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

Example:

- R ↦ (R x R) / S_2 associates to any monad R the module of its unordered pairs. Models (in particular the syntax) are monads equipped with a binary *commutative* operation.

Induction

Example: computing the free variables of a lambda-term

We compute it by induction on the syntax:

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

This is formalized in our setting as a family of maps *(*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:

$$\begin{tikzcd} X \arrow[r, "var_{L,X}"] \arrow[d, equal] & L(X) \arrow[d, dashed, "fv_X"] \\ X \arrow[r, "var_{\mathscr{P},X}"'] & \mathscr{P}(X) \end{tikzcd}
\qquad
\begin{tikzcd} L(X) \arrow[r, "< t >_L"] \arrow[d, dashed, "fv_X"'] & L(Y) \arrow[d, dashed, "fv_Y"] \\ \mathscr{P}(X) \arrow[r, "< t >_{\mathscr{P}}"'] & \mathscr{P}(Y) \end{tikzcd}$$

$$\begin{tikzcd} L(X) \times L(X) \arrow[r, "app_L"] \arrow[d, dashed, "fv_X \times fv_X"'] & L(X) \arrow[d, dashed, "fv_X"] \\ \mathscr{P}(X) \times \mathscr{P}(X) \arrow[r, "app_{\mathscr{P}}"'] & \mathscr{P}(X) \end{tikzcd}
\qquad
\begin{tikzcd} L(X+\{n\}) \arrow[r, "abs_L"] \arrow[d, dashed, "fv_{X+\{n\}}"'] & L(X) \arrow[d, dashed, "fv_X"] \\ \mathscr{P}(X+\{n\}) \arrow[r, "abs_{\mathscr{P}}"'] & \mathscr{P}(X) \end{tikzcd}$$

Induction and initiality

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

$$X \xrightarrow[\mathrm{var}_{R,X}]{} R(X)$$

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

$$R(X) \times R(X) \xrightarrow[\mathrm{app}_R]{} R(X)$$

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

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:

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

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

(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 $\boldsymbol{\sigma}_R$: $\boldsymbol{\Sigma}_{\mathbf{R}}$ → **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 → 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 → 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 → 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

Quotient of a signature

Quotient of a set:

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

(x ~ x' iff f(x) = f(y)).

Quotient of a signature:

A quotient of a signature $\mathbf{\Sigma}$ is a signature $\mathbf{\Psi}$ together with a family of module morphisms $(\mathbf{f_R} : \mathbf{\Sigma_R} \to \mathbf{\Psi_R})_{\mathbf{R}}$ that is pointwise surjective and commutes with any monad morphism $\mathbf{m} : \mathbf{R} \to \mathbf{R'}$ in the sense that:

$$f_{R'} \circ \Sigma(m) = \Psi(m) \circ f_R \qquad \text{(naturality condition)}$$

Quotient of a signature

Quotient of a set:

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

(x ~ x' iff f(x) = f(y)).

Quotient of a signature:

A quotient of a signature **Σ** is a signature **Ψ** together with a family of module morphisms $(\mathbf{f}_\mathbf{R} : \mathbf{\Sigma}_\mathbf{R} \to \mathbf{\Psi}_\mathbf{R})_\mathbf{R}$ that is pointwise surjective and commutes with any monad morphism **m** : **R** → **R'** in the sense that:

$$f_{R'} \circ \Sigma(m) = \Psi(m) \circ f_R \qquad \text{(naturality condition)}$$

Quotients of algebraic signatures

Theorem: Let **S** be the syntax generated by an algebraic signature **Σ**. Then any quotient **Ψ** of **Σ** generates a syntax (obtained by quotienting adequatly the syntax **S**)

Examples of quotient algebraic signatures:
TODO

Quotients of algebraic signatures

Theorem: Let **S** be the syntax generated by an algebraic signature $\boldsymbol{\Sigma}$. Then any quotient $\boldsymbol{\Psi}$ of $\boldsymbol{\Sigma}$ generates a syntax (obtained by quotienting adequatly the syntax **S**)

Examples of quotient algebraic signatures:

TODO