# High-level signatures and initial semantics

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

# Language with substitutions

Language with substitutions

**Goal of our work:** give a mathematical account of "languages" with *variables* and *substitution* (i.e. replacing variables with any expression yields a valid expression).

**Aim:** justify induction on the syntax with an *initiality* property

**First Example:** formal arithmetic expressions with +, ×, natural numbers.

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

# PLAN

**1. Languages, monads and modules**

2. Induction and Initiality

3. Signatures

# Languages as monads

**The language of arithmetic expressions as a monad:**

- for any set of variables *X={x,y,z,..}*, there is a set *A(X)* of expressions taking free variables in *X*.

$$A(\emptyset) = \{0, \quad 2, \quad 1+2, \quad 5 \times (3+1), \dots\}$$

$$A(\{x\}) = A(\emptyset) \cup \{x, \quad x+3, \quad 5 \times x, \dots\}$$

- any variable $x \in X$ is a valid expression that we note $\underline{x} \in A(X)$

$$\forall X,\ var_X : X \to A(X)$$

$$x \mapsto \underline{x}$$

- given a family $(t_x)_{x \in X}$ of expressions in *A(Y)*, we can substitute each variable *x* of an expression $e \in A(X)$ with $t_x$:

$$< t >: A(X) \to A(Y)$$

$$e \mapsto e[x \mapsto t_x]$$

# 1st monad law

**The identity substitution:**

Substituting all variables with the same variables (i.e. according to the family $(var_X(x))_{x \in X}$) does nothing:

$$\forall e \in A(X),\ e[x \mapsto x] = e$$

In other words, the following diagram commutes:

$$\begin{tikzcd} A(X) \arrow[r, bend left, "<var_X>"] \arrow[r, bend right, "id_X"'] & A(X) \end{tikzcd}$$

# 2nd monad law

**Substitution of a variable:**

Let $(t_x)_{x \in X}$ be a family of expressions in A(Y):

$$t : X \to A(Y)$$

Substituting an expression consisting of a single variable *var(y)* yields $t_y$:

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

In other words, the following diagram commutes:

$$\begin{tikzcd}
X \arrow[r, "var_X"] \arrow[dr, "t"'] & A(X) \arrow[d, "\langle t \rangle"] \\
& A(Y)
\end{tikzcd}$$

# 3rd monad law

**Composition of substitutions:**

Let

- $(t_x)_{x \in X}$ be a family of expressions in A(Y) $\quad t : X \to A(Y)$

- $(u_y)_{y \in Y}$ be a family of expressions in A(Z) $\quad u : Y \to A(Z)$

Then, for any expression e in A(X),

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

In other words, the following diagram commutes:

$$\begin{tikzcd}
A(X) \arrow[d, "<t>"'] \arrow[dr, "< <u> \circ t >"] & \\
A(Y) \arrow[r, "<u>"'] & A(Z)
\end{tikzcd}$$

# Languages as monads

**The language of arithmetic expressions as a monad:**

We have, for each set X, a set A(X), and maps:

$$\forall X,\ var_X : X \to A(X)$$

$$\forall X, Y, (t_x \in A(Y))_{x \in X},\ < t >: A(X) \to A(Y)$$

subjected to the three laws:

$$\begin{tikzcd} A(X) \arrow[r, bend left, "< var_X >"] \arrow[r, bend right, "id_X"'] & A(X) \end{tikzcd}$$

IDENTITY SUBSTITUTION

$$\begin{tikzcd} X \arrow[r, "var_X"] \arrow[dr, "t"'] & A(X) \arrow[d, "< t >"] \\ & A(Y) \end{tikzcd}$$

VARIABLE SUBSTITUTION

$$\begin{tikzcd} A(X) \arrow[d, "< t >"'] \arrow[dr, "< <u> \circ t >"] & \\ A(Y) \arrow[r, "<u>"'] & A(Z) \end{tikzcd}$$

COMPOSITION OF

SUBSTITUTIONS

This is the definition of a monad on the category of Sets

# Examples of monads

Examples of monads

**Some other examples of monads:**

- 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(∅)

$$(\lambda x.x{\color{blue}z})[z \mapsto \lambda y.y] = \lambda x.x{\color{blue}(\lambda y.y)}$$

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

# Examples of monads

Examples of monads

**Some other examples of monads:**

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

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

Let U ⊂ X (i.e. U ∈ $\mathscr{P}$(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:

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

We characterize this situation as follows:

*A(X) x A(X)* has a notion of substitution $\rightsquigarrow$ A x A is a ***module*** on A

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

# Module over a monad

**Substitution on A x A:**

Let $(t_x)_{x \in X}$ be a family of expressions in A(Y): $t : X \to A(Y)$

Then we can define substitution on *A(X) x A(X)*:

$$< t >: A(X) \times A(X) \to A(Y) \times A(Y)$$
$$(e, e') \mapsto (e, e')[x \mapsto t_x] \quad := (e[x \mapsto t_x], e'[x \mapsto t_x])$$

that inherit some properties of substitution on *A*:

- **(identity substitution)** $(e, e')[x \mapsto x] = (e, e')$
- **(composition of substitutions)** for any other family $(u_y)_{y \in Y}$ of expressions in *A(Z)*, $(e,e')[x \mapsto t_x][y \mapsto u_y] = (e,e')[x \mapsto t_x[y \mapsto u_y]_A]$

This is an example of a module over the monad A

# Module over a monad

**Module over a monad:**

A module over the monad A:

- associates a set M(X) to any set X: M(X) can be thought of as "generalized" expressions taking variables in X.
- is equipped, given any family $(t_x)_{x \in X}$ of elements of A(Y), with a substitution $< t >_M : M(X) \rightarrow M(Y)$ satisfying:

$$
\begin{tikzcd}
M(X) \arrow[r, bend left, "<var_X>_M"] \arrow[r, bend right, "id_X"'] & M(X)
\end{tikzcd}
$$

IDENTITY SUBSTITUTION

$$
\begin{tikzcd}
M(X) \arrow[d, "<t>_M"'] \arrow[dr, "< <u>_A \circ t >_M"] & \\
M(Y) \arrow[r, "<u>_M"'] & M(Z)
\end{tikzcd}
$$

COMPOSITION OF SUBSTITUTIONS

# Examples of modules

**Modules over a monad:**

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

- **R** itself (already satisfies identity substitution and composition of substitution by definition of a monad)
- **R x R** (i.e. the assignement $X \mapsto R(X) \times R(X)$)
- **M x N** for any module M and N

**Important example : Derivative of a module**

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

# Module morphism

**Module morphism:**

Let **M** and **N** be two modules over a monad **R**. A module morphism between **M** and **N** is a family of maps $(f_X\!:\!M(X) \to N(X))_X$ that *commutes with substitution*: for any $e \in M(X)$ and family $(t_x)_{x \in X}$ of elements of *M(Y)*,

$$f_X(e)[x \mapsto t_x]_N = f_Y(e[x \mapsto y_x]_M)$$

$$\begin{tikzcd} M(X) \arrow[r, "< t >_M"] \arrow[d, "f_X"'] & M(Y) \arrow[d, "f_Y"] \\ N(X) \arrow[r, "< t >_N"'] & N(Y) \end{tikzcd}$$

**Example:**

$$add : A \times A \to A$$

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

# Examples of module morphisms

**Some module morphisms:**

- $\mathbf{id_M : M \to M}$ 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: $app(t,u) = t\ u$

- What about the abstraction operation abs : t ↦ λx.t of lambda calculus?

**Binding variables:**

In $\lambda x.t$, the term $t$ depends on an additional free variable $x$:
If $\lambda x.t \in L(Y)$, then $t \in L(Y + \{x\})$ $\mathbf{= L'(Y)}$

## $\mathbf{abs:L' \to L}$ is a module morphism

# PLAN

1. Languages, monads and modules

**2. Induction and Initiality**

3. Signatures

# Induction

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

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:

$$\begin{tikzcd} X \arrow[r, "\mathrm{var}_{L,X}"] \arrow[d, equal] & L(X) \arrow[d, dashed, "\mathrm{fv}_X"] \\ X \arrow[r, "\mathrm{var}_{\mathscr{P},X}"'] & \mathscr{P}(X) \end{tikzcd}$$

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

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

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

# Induction and initiality

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

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

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

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

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

# 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:

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

$$
\begin{tikzcd}
L(X) \times L(X) \arrow[r, "\mathrm{app}_L"] \arrow[d, dashed, "f_X \times f_X"'] & L(X) \arrow[d, dashed, "f_X"] \\
R(X) \times R(X) \arrow[r, "\mathrm{app}_R"'] & R(X)
\end{tikzcd}
\qquad
\begin{tikzcd}
L(X+\{n\}) \arrow[r, "\mathrm{abs}_L"] \arrow[d, dashed, "f_{X+\{n\}}"'] & L(X) \arrow[d, dashed, "f_X"] \\
R(X+\{n\}) \arrow[r, "\mathrm{abs}_R"'] & R(X)
\end{tikzcd}
$$

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

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

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

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

# 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

1. Languages, monads and modules

2. Induction and Initiality

**3. Signatures**

# Definition of 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 $R \times R \to R$.

  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

# 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 $R \mapsto R' \times R'' \times R''' + R \times R'' \times R''' \times R + \ldots$ (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.

# 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 $(\mathbf{f_R} : \mathbf{\Sigma_R} \to \mathbf{\Psi_R})_\mathbf{R}$ that is pointwise surjective and commutes with any monad morphism **m** : **R** → **R'** in the sense that:

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

# 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 ($\mathbf{f_R} : \mathbf{\Sigma_R} \to \mathbf{\Psi_R})_{\mathbf{R}}$ 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 an algebraic signatures

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

**Examples of quotient algebraic signatures:**

TODO

# Quotients of an 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