

Higher-order Arities, Signatures and Equations via Modules

Ambroise Lafont

joint work with
Benedikt Ahrens, André Hirschowitz, Marco Maggesi

Keywords associated with syntax

Induction/Recursion

Substitution

Model

Syntax

Operation/Construction

Arity/Signature

This talk: give a *discipline* for specifying syntaxes

Motivating example: dLC

syntax of dLC = **differential λ -calculus** [Ehrhard-Regnier 2003].

- explicitly involves **equations** e.g. $s+t = t+s$
- specifically tailored: (not an *instance* of a general framework/scheme)
inductive definition of a set + ad-hoc structure
e.g. **unary substitution**

Our proposal = a discipline for presenting syntaxes

- signature = operations + equations
- [Fiore-Hure 2010]: alternative approach, for simply typed syntaxes
 \Rightarrow our approach explicitly relies on monads and modules (untyped case).

Syntax of dLC: [Ehrhard-Regnier 2003]

Let be given a denumerable set of variables. We define by induction on k an increasing family of sets (Δ_k) . We set $\Delta_0 = \emptyset$ and Δ_{k+1} is defined as follows.

Monotonicity: if t belongs to Δ_k then t belongs to Δ_{k+1} .

Variable: if $n \in \mathbb{N}$, x is a variable, $i_1, \dots, i_n \in \mathbb{N}^+ = \mathbb{N} \setminus \{0\}$ and $u_1, \dots, u_n \in \Delta_k$, then

$$D_{i_1, \dots, i_n} x \cdot (u_1, \dots, u_n)$$

belongs to Δ_{k+1} . This term is identified with all the terms of the shape $D_{i_{\sigma(1)}, \dots, i_{\sigma(n)}} x \cdot (u_{\sigma(1)}, \dots, u_{\sigma(n)}) \in \Delta_{k+1}$ where σ is a permutation on $\{1, \dots, n\}$.

Abstraction: if $n \in \mathbb{N}$, x is a variable, $u_1, \dots, u_n \in \Delta_k$ and $t \in \Delta_k$, then

$$D_1^n \lambda x t \cdot (u_1, \dots, u_n)$$

belongs to Δ_{k+1} . This term is identified with all the terms of the shape $D_1^n \lambda x t \cdot (u_{\sigma(1)}, \dots, u_{\sigma(n)}) \in \Delta_{k+1}$ where σ is a permutation on $\{1, \dots, n\}$.

Application: if $s \in \Delta_k$ and $t \in R\langle \Delta_k \rangle$, then

$$(s)t$$

belongs to Δ_{k+1} .

Setting $n=0$ in the first two clauses, and restricting application by the constraint that $t \in \Delta_k \subseteq R\langle \Delta_k \rangle$, one retrieves the usual definition of lambda-terms which shows that differential terms are a superset of ordinary lambda-terms.

The permutative identification mentioned above will be called *equality up to differential permutation*. We also work up to α -conversion.

Syntax of dLC: [Ehrhard-Regnier 2003]

Let be given a denumerable set of variables. We define by induction on k an increasing family of sets (Δ_k) . We set $\Delta_0 = \emptyset$ and Δ_{k+1} is defined as follows.

Monotonicity: if t belongs to Δ_k then t belongs to Δ_{k+1} .

Variable: if $n \in \mathbb{N}$, x is a variable, $i_1, \dots, i_n \in \mathbb{N}^+ = \mathbb{N} \setminus \{0\}$ and $u_1, \dots, u_n \in \Delta_k$, then

$$D_{i_1, \dots, i_n} x \cdot (u_1, \dots, u_n)$$

belongs to Δ_{k+1} . This term is identified with all the terms of the shape $D_{i_{\sigma(1)}, \dots, i_{\sigma(n)}} x \cdot (u_{\sigma(1)}, \dots, u_{\sigma(n)}) \in \Delta_{k+1}$ where σ is a permutation on $\{1, \dots, n\}$.

Abstraction: if $n \in \mathbb{N}$, x is a variable, $u_1, \dots, u_n \in \Delta_k$ and $t \in \Delta_k$, then

$$D_1^n \lambda x t \cdot (u_1, \dots, u_n)$$

belongs to Δ_{k+1} . This term is identified with all the terms of the shape $D_1^n \lambda x t \cdot (u_{\sigma(1)}, \dots, u_{\sigma(n)}) \in \Delta_{k+1}$ where σ is a permutation on $\{1, \dots, n\}$.

Application: if $s \in \Delta_k$ and $t \in R\langle \Delta_k \rangle$, then

$$(s)t \leftarrow \text{as an operation: } \Lambda \times \text{FreeCommutativeMonoid}(\Lambda) \rightarrow \Lambda$$

belongs to Δ_{k+1} .

Setting $n=0$ in the first two clauses, and restricting application by the constraint that $t \in \Delta_k \subseteq R\langle \Delta_k \rangle$, one retrieves the usual definition of lambda-terms which shows that differential terms are a superset of ordinary lambda-terms.

The permutative identification mentioned above will be called equality up to differential permutation. We also work up to α -conversion.

Syntax of dLC: [BEM 2010]

A **syntax** for the **differential λ -calculus** by **mutual induction**:

[Bucciarelli-Ehrhard-Manzonetto 2010]

Simple terms:

$$\Lambda^s : \quad s, t \quad ::= \quad x \mid \lambda x. s \mid sT \mid D s \cdot t$$

Differential λ -terms:


$$\Lambda^d : \quad T \quad ::= \quad 0 \mid s \mid s + T$$

Syntax of dLC: [BEM 2010]



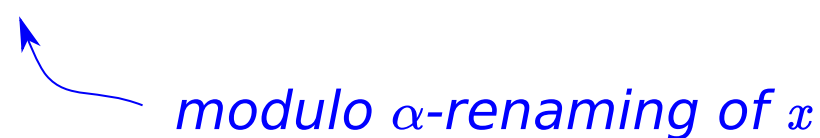
A **syntax** for the **differential λ -calculus** by **mutual induction**:

[Bucciarelli-Ehrhard-Manzonetto 2010]

Simple terms:

$$\Lambda^s : \quad s, t \quad ::= \quad x \mid \lambda x. s \mid sT \mid Ds \cdot t$$


Differential λ -terms:


$$\Lambda^d : \quad T \quad ::= \quad 0 \mid s \mid s + T$$


Syntax of dLC: [BEM 2010]



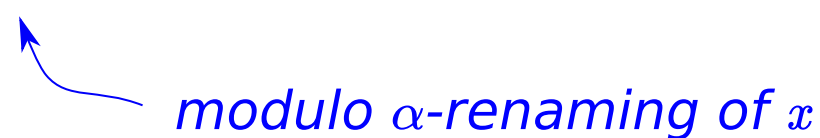
A **syntax** for the **differential λ -calculus** by **mutual induction**:

[Bucciarelli-Ehrhard-Manzonetto 2010]

Simple terms:

$$\Lambda^s : \quad s, t \quad ::= \quad x \mid \lambda x. s \mid sT \mid Ds \cdot t$$


Differential λ -terms:

$$\Lambda^d : \quad T \quad ::= \quad 0 \mid s \mid s + T$$



$$\Lambda^d = \mathbf{FreeCommutativeMonoid}(\Lambda^s)$$

Syntax of dLC: [BEM 2010]

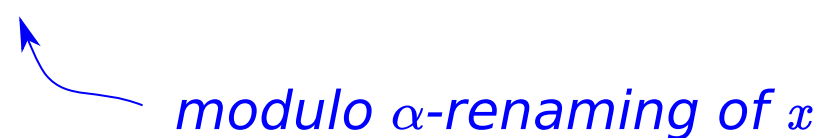
A **syntax** for the **differential λ -calculus** by **mutual induction**:

[Bucciarelli-Ehrhard-Manzonetto 2010]

Simple terms:

$$\Lambda^s : \quad s, t \quad ::= \quad x \mid \lambda x. s \mid sT \mid Ds \cdot t$$


Differential λ -terms:

$$\Lambda^d : \quad T \quad ::= \quad 0 \mid s \mid s + T$$


modulo α -renaming of x

neutral element for $+$

modulo commutativity

$$\Lambda^d = \mathbf{FreeCommutativeMonoid}(\Lambda^s)$$

Syntax: specified by operations and **equations**.

Syntax of dLC: [BEM 2010]

A **syntax** for the **differential λ -calculus** by **mutual induction**:

[Bucciarelli-Ehrhard-Manzonetto 2010]

Simple terms:

$$\Lambda^s : \quad s, t \quad ::= \quad x \mid \lambda x. s \mid sT \mid Ds \cdot t$$

variable (arrow from x to $\lambda x. s$)

Differential λ -terms:

$$\Lambda^d : \quad T \quad ::= \quad 0 \mid s \mid s + T$$

modulo α -renaming of x (arrow from $\lambda x. s$ to s)

neutral element for $+$ (arrow from 0 to $s + T$)

modulo commutativity (arrow from $s + T$ to $s + T$)

$$\Lambda^d = \mathbf{FreeCommutativeMonoid}(\Lambda^s)$$

Syntax: specified by operations and **equations**.

But which ones are allowed ? What is the limit ?

Syntax of dLC: Our version

Which operations/equations are allowed to specify a syntax ?

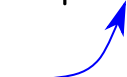

A stand-alone presentation of differential λ -terms:

Allow sums everywhere (not only in the right arg of application)

Differential λ -terms:

$$\Lambda^d : S, T ::= x \mid \lambda x. S \mid S T \mid D S \cdot T$$

$$\mid 0 \mid S + T$$

neutral element for +   *modulo commutativity and associativity*

Macros in [BEM 2010]:

$$\lambda x. \Sigma_i t_i := \Sigma_i \lambda x. t_i$$

$$(\Sigma_i t_i) u := \Sigma_i t_i u$$

$$D(\Sigma_i t_i) \cdot (\Sigma_j u_j) := \Sigma_i \Sigma_j D t_i \cdot u_j$$

Syntax of dLC: Conclusion

How can we compare these different versions ?

In which sense are they syntaxes ?

Which operations/equations are we allowed to specify in a syntax ?

Syntax of dLC: Conclusion

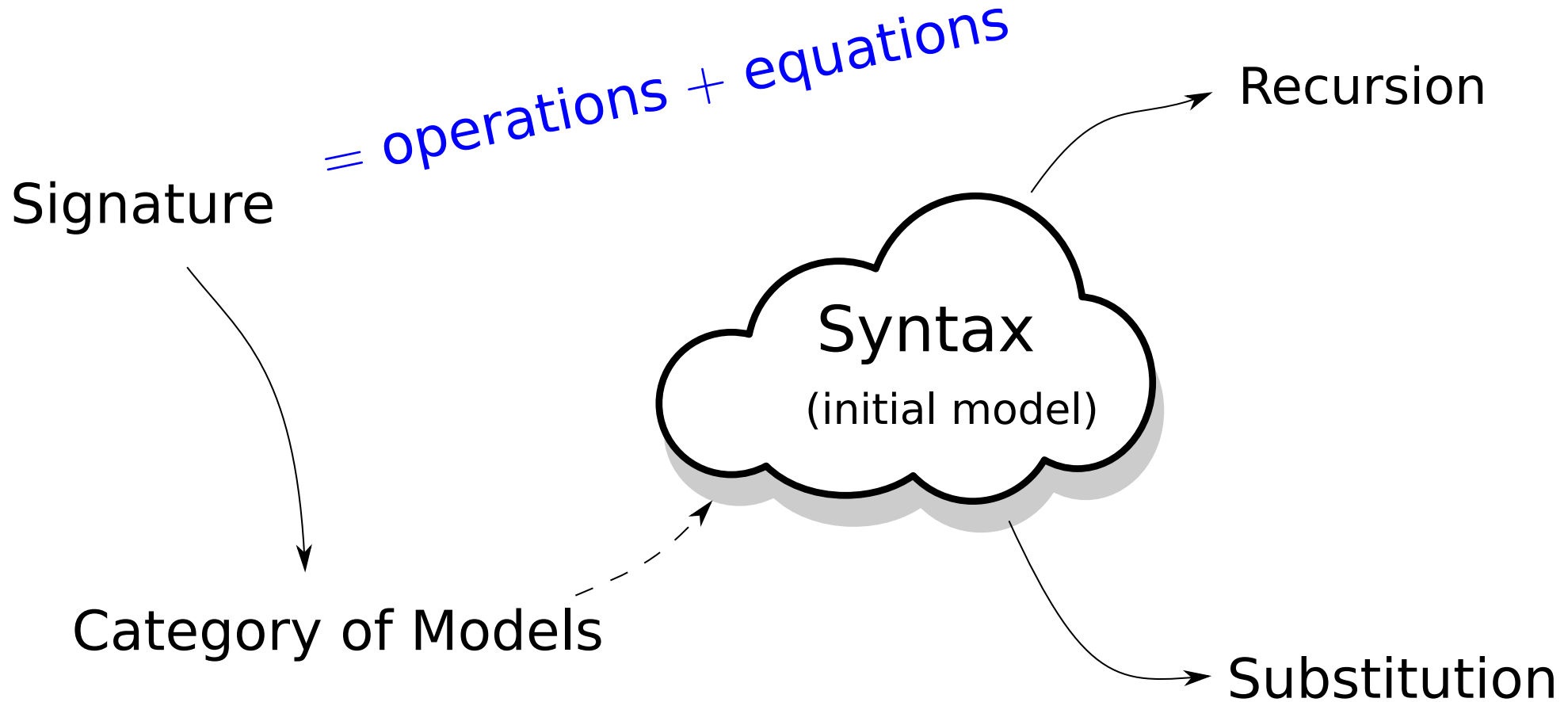
How can we compare these different versions ?

In which sense are they syntaxes ?

Which operations/equations are we allowed to specify in a syntax ?

What is a syntax ?

What is a syntax?



generates a syntax = existence of the initial model

Table of contents

1. 1-Signatures and models based on monads and modules

2. Equations

3. Recursion

Table of contents

1. 1-Signatures and models based on monads and modules

- Substitution and monads
- 1-Signatures and their models

2. Equations

3. Recursion

Substitution and monads

Example: differential λ -calculus

$$\Lambda^d : S, T ::= x \mid \lambda x. S \mid S T \mid DS \cdot T \\ \mid 0 \mid S + T$$

Free variable indexing:

$$dLC : X \mapsto \{\text{terms taking free variables in } X\}$$

$$dLC(\emptyset) = \{0, \lambda z.z, \dots\}$$

$$dLC(\{x, y\}) = \{0, \lambda z.z, \dots, x, y, x + y, \dots\}$$

Substitution and monads

Example: differential λ -calculus

$$\Lambda^d : S, T ::= x \mid \lambda x. S \mid S T \mid DS \cdot T \\ \mid 0 \mid S + T$$

Free variable indexing:

$$dLC : X \mapsto \{\text{terms taking free variables in } X\}$$

$$dLC(\emptyset) = \{0, \lambda z.z, \dots\}$$

$$dLC(\{x, y\}) = \{0, \lambda z.z, \dots, x, y, x + y, \dots\}$$

Parallel substitution:

Substitution and monads

Example: differential λ -calculus

$$\Lambda^d : \quad S, T \quad ::= \quad x \mid \lambda x. S \mid S T \mid DS \cdot T \\ \mid 0 \mid S + T$$

Free variable indexing:

$$dLC : X \mapsto \{\text{terms taking free variables in } X\}$$

$$dLC(\emptyset) = \{0, \lambda z.z, \dots\}$$

$$dLC(\{x, y\}) = \{0, \lambda z.z, \dots, x, y, x + y, \dots\}$$

Parallel substitution:

$$\begin{array}{ll} \text{bind}_f : dLC(X) & \rightarrow dLC(Y) \\ t & \mapsto t[x \mapsto f(x)] \end{array} \quad \text{where } f : X \rightarrow dLC(Y)$$

$\Rightarrow (dLC, \text{var}_X : X \subset dLC(X), \text{bind}) = \mathbf{monad \ on \ Set}$

Substitution and monads

Example: differential λ -calculus

$$\Lambda^d : \quad S, T \quad ::= \quad x \mid \lambda x. S \mid S T \mid DS \cdot T \\ \mid 0 \mid S + T$$

Free variable indexing:

$$dLC : X \mapsto \{\text{terms taking free variables in } X\}$$

$$dLC(\emptyset) = \{0, \lambda z.z, \dots\}$$

$$dLC(\{x, y\}) = \{0, \lambda z.z, \dots, x, y, x + y, \dots\}$$

Parallel substitution:

$$\begin{array}{ll} \text{bind}_f : dLC(X) \rightarrow dLC(Y) & \text{where } f : X \rightarrow dLC(Y) \\ t \mapsto t[x \mapsto f(x)] \end{array}$$

$\Rightarrow (dLC, \text{var}_X : X \subset dLC(X), \text{bind}) = \mathbf{monad\ on\ Set}$

monad morphism = mapping preserving variables and substitutions.

Preview: Operations are module morphisms

+ commutes with substitution

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

Categorical formulation

$dLC \times dLC$ supports
 dLC -substitution



$dLC \times dLC$ is a **module over dLC**

+ commutes
with substitution



$+ : dLC \times dLC \rightarrow dLC$ is a
module morphism

Building blocks for specifying operations

Essential constructions of **modules over a monad** R :

- R itself

- $M \times N$ for any modules M and N

e.g. $R \times R$:

$$(t, u)[x \mapsto f(x)] := (t[x \mapsto f(x)], u[x \mapsto f(x)]) \quad \text{where } f: X \rightarrow R(Y)$$

- $M' = \text{derivative of a module } M$: $M'(X) = M(X \amalg \{\diamond\})$.

used to model an operation binding a variable (Cf next slide).

Syntactic operations are module morphisms

operations = module morphisms = maps commuting with substitution.

$$\text{app} : \text{dLC} \times \text{dLC} \rightarrow \text{dLC}$$

$$0 : 1 \rightarrow \text{dLC}$$

$$\text{abs} : \text{dLC}' \rightarrow \text{dLC}$$

$$+ : \text{dLC} \times \text{dLC} \rightarrow \text{dLC}$$

$$\text{abs}_X : \text{dLC}(X \coprod \{\diamond\}) \rightarrow \text{dLC}(X)$$
$$t \mapsto \lambda \diamond. t$$

Syntactic operations are module morphisms

operations = module morphisms = maps commuting with substitution.

$$\text{app} : \text{dLC} \times \text{dLC} \rightarrow \text{dLC}$$

$$0 : 1 \rightarrow \text{dLC}$$

$$\text{abs} : \text{dLC}' \rightarrow \text{dLC}$$

$$+ : \text{dLC} \times \text{dLC} \rightarrow \text{dLC}$$

$$\text{abs}_X : \text{dLC}(X \coprod \{\diamond\}) \rightarrow \text{dLC}(X)$$
$$t \mapsto \lambda \diamond. t$$

Combining operations into a single one using disjoint union

$$[\text{app}, \text{abs}] : (\text{dLC} \times \text{dLC}) \coprod \text{dLC}' \rightarrow \text{dLC}$$

$$[0, +] : 1 \coprod (\text{dLC} \times \text{dLC}) \rightarrow \text{dLC}$$

1-signatures and their models

A **1-signature** Σ = functorial assignment:

$$R \mapsto \Sigma(R)$$

Example: $(0, +)$

$$\Sigma_{0,+}(R) = 1 \coprod (R \times R)$$

A **model of Σ** is a pair:

$$(R, \rho : \Sigma(R) \rightarrow R)$$

dLC = model of $\Sigma_{0,+}$

$$[0, +] : 1 \coprod (dLC \times dLC) \rightarrow dLC$$

A **model morphism** $m : (R, \rho) \rightarrow (S, \sigma)$ = monad morphism commuting with the module morphism:

$$\begin{array}{ccc} \Sigma(R) & \xrightarrow{\rho} & R \\ \Sigma(m) \downarrow & & \downarrow m \\ \Sigma(S) & \xrightarrow{\sigma} & S \end{array}$$

1-signatures and their models

A **1-signature** Σ = functorial assignment:

$$R \mapsto \Sigma(R)$$

monad

A **model of Σ** is a pair:

$$(R, \rho : \Sigma(R) \rightarrow R)$$

Example: $(0, +)$

$$\Sigma_{0,+}(R) = 1 \coprod (R \times R)$$

dLC = model of $\Sigma_{0,+}$

$$[0, +] : 1 \coprod (dLC \times dLC) \rightarrow dLC$$

A **model morphism** $m : (R, \rho) \rightarrow (S, \sigma) =$ monad morphism commuting with the module morphism:

$$\begin{array}{ccc} \Sigma(R) & \xrightarrow{\rho} & R \\ \Sigma(m) \downarrow & & \downarrow m \\ \Sigma(S) & \xrightarrow{\sigma} & S \end{array}$$

1-signatures and their models

A **1-signature** Σ = functorial assignment:

$$R \mapsto \Sigma(R)$$

monad \quad module over R

A **model of Σ** is a pair:

$$(R, \rho : \Sigma(R) \rightarrow R)$$

Example: $(0, +)$

$$\Sigma_{0,+}(R) = 1 \coprod (R \times R)$$

dLC = model of $\Sigma_{0,+}$

$$[0, +] : 1 \coprod (dLC \times dLC) \rightarrow dLC$$

A **model morphism** $m : (R, \rho) \rightarrow (S, \sigma)$ = monad morphism commuting with the module morphism:

$$\begin{array}{ccc} \Sigma(R) & \xrightarrow{\rho} & R \\ \Sigma(m) \downarrow & & \downarrow m \\ \Sigma(S) & \xrightarrow{\sigma} & S \end{array}$$

1-signatures and their models

A **1-signature** Σ = functorial assignment:

$$R \mapsto \Sigma(R)$$

monad \nearrow \nwarrow module over R

A **model of Σ** is a pair:

$$(R, \rho : \Sigma(R) \rightarrow R)$$

monad \nearrow

Example: $(0, +)$

$$\Sigma_{0,+}(R) = 1 \coprod (R \times R)$$

dLC = model of $\Sigma_{0,+}$

$$[0, +] : 1 \coprod (dLC \times dLC) \rightarrow dLC$$

A **model morphism** $m : (R, \rho) \rightarrow (S, \sigma) =$ monad morphism commuting with the module morphism:

$$\begin{array}{ccc} \Sigma(R) & \xrightarrow{\rho} & R \\ \Sigma(m) \downarrow & & \downarrow m \\ \Sigma(S) & \xrightarrow{\sigma} & S \end{array}$$

1-signatures and their models

A **1-signature** Σ = functorial assignment:

$$R \mapsto \Sigma(R)$$

monad \nearrow \nwarrow module over R

A **model of Σ** is a pair:

$$(R, \rho : \Sigma(R) \rightarrow R)$$

monad \nearrow \nwarrow module morphism

Example: $(0, +)$

$$\Sigma_{0,+}(R) = 1 \coprod (R \times R)$$

dLC = model of $\Sigma_{0,+}$

$$[0, +] : 1 \coprod (dLC \times dLC) \rightarrow dLC$$

A **model morphism** $m : (R, \rho) \rightarrow (S, \sigma)$ = monad morphism commuting with the module morphism:

$$\begin{array}{ccc} \Sigma(R) & \xrightarrow{\rho} & R \\ \Sigma(m) \downarrow & & \downarrow m \\ \Sigma(S) & \xrightarrow{\sigma} & S \end{array}$$

Syntax

Definition

Given a 1-signature Σ , its **syntax** is an initial object in its category of models.

Question: Does the syntax exist for every 1-signature?

Answer: No.

Syntax

Definition

Given a 1-signature Σ , its **syntax** is an initial object in its category of models.

Question: Does the syntax exist for every 1-signature?

Answer: No.

Counter-example: The 1-signature $R \mapsto \mathcal{P} \circ R$ has a syntax S .

powerset endofunctor on Set



Examples of 1-signatures generating syntax

- **(0,+) language:**

Signature: $R \mapsto 1 \coprod (R \times R)$

Model: $(R, \quad 0 : 1 \rightarrow R, \quad + : R \times R \rightarrow R)$

Syntax: $(B, \quad 0 : 1 \rightarrow B, \quad + : B \times B \rightarrow B)$

- **lambda calculus:**

Signature: $R \mapsto R' \coprod (R \times R)$

Model: $(R, \quad abs : R' \rightarrow R, \quad app : R \times R \rightarrow R)$

Syntax: $(\Lambda, \quad abs : \Lambda' \rightarrow \Lambda, \quad app : \Lambda \times \Lambda \rightarrow \Lambda)$

Can we generalize this pattern?

Initial semantics for algebraic 1-signatures

Theorem [Hirschowitz & Maggesi 2007]

Syntax exists for any **algebraic 1-signature**, i.e. 1-signature built out of derivatives, products, disjoint unions, and the 1-signature $R \mapsto R$.

Algebraic 1-signatures correspond to the binding signatures described in [Fiore-Plotkin-Turi 1999]

(binding signatures = lists of natural numbers specify n-ary operations, possibly binding variables)

Question: Can we enforce some equations in the syntax ?

e.g. **associativity** and **commutativity** of $+$ for the differential λ -calculus.

Quotients of algebraic 1-signatures

More sophisticated 1-signatures: ***quotients*** of algebraic 1-signatures.

Theorem [AHLM CSL 2018]

Syntax exists for any "***quotient***" of algebraic 1-signature.

Examples:

- a **commutative** binary operation
- application of the simple terms of differential λ -calculus (2nd variant)

app : $\text{dLC} \times \text{FreeCommutativeMonoid}(\text{dLC}) \rightarrow \text{dLC}$

Quotients of algebraic 1-signatures

More sophisticated 1-signatures: ***quotients*** of algebraic 1-signatures.

Theorem [AHLM CSL 2018]

Syntax exists for any "***quotient***" of algebraic 1-signature.

Examples:

- a **commutative** binary operation
- application of the simple terms of differential λ -calculus (2nd variant)

app : $\text{dLC} \times \text{FreeCommutativeMonoid}(\text{dLC}) \rightarrow \text{dLC}$

... but not enough for the differential λ -calculus:

- **associativity** of $+$
- **linearity** of the operations

Table of contents

1. 1-Signatures and models based on monads and modules

2. Equations

3. Recursion

Example: a commutative binary operation

Specification of a binary operation

1-Signature: $R \mapsto R \times R$

Model: $(R, + : R \times R \rightarrow R)$

What is an appropriate notion of model for a commutative binary operation ?

Example: a commutative binary operation

Specification of a **commutative** binary operation

1-Signature: $R \mapsto R \times R$

Model: $(R, + : R \times R \rightarrow R)$ s.t. $t + u = u + t$ (1)

What is an appropriate notion of model for a commutative binary operation ?

Answer: a monad equipped with a **commutative** binary operation

Example: a commutative binary operation

Specification of a **commutative** binary operation

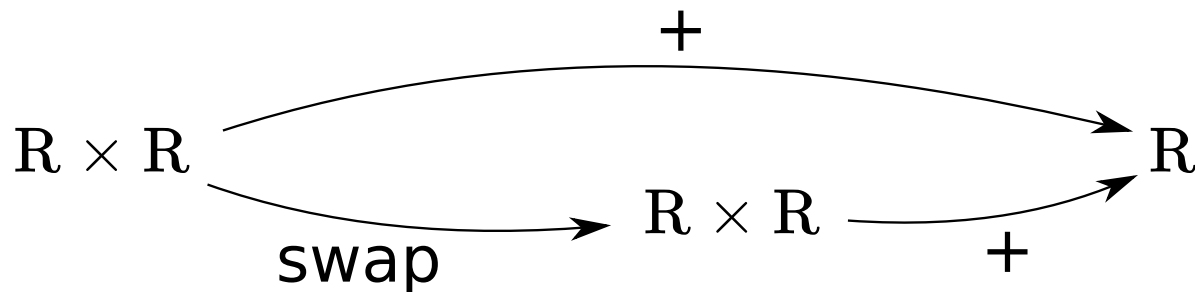
1-Signature: $R \mapsto R \times R$

Model: $(R, + : R \times R \rightarrow R)$ s.t. $t + u = u + t$ (1)

What is an appropriate notion of model for a commutative binary operation ?

Answer: a monad equipped with a **commutative** binary operation

Equation (1) states an equality between R -module morphisms:



Equations

Given a 1-signature Σ , (e.g. binary operation: $\Sigma(R) = R \times R$)

a Σ -**equation** $A \rightrightarrows B$ is a functorial assignment: e.g. commutativity:

$$R \mapsto \left(A(R) \rightrightarrows B(R) \right)$$

model of Σ (points to R)

parallel pair of module morphisms over R (points to $A(R) \rightrightarrows B(R)$)

$$R \mapsto \left(R \times R \xrightarrow[+\circ swap]{+} R \right)$$

A **2-signature** is a pair

$$(\Sigma, E)$$

1-signature (points to Σ)

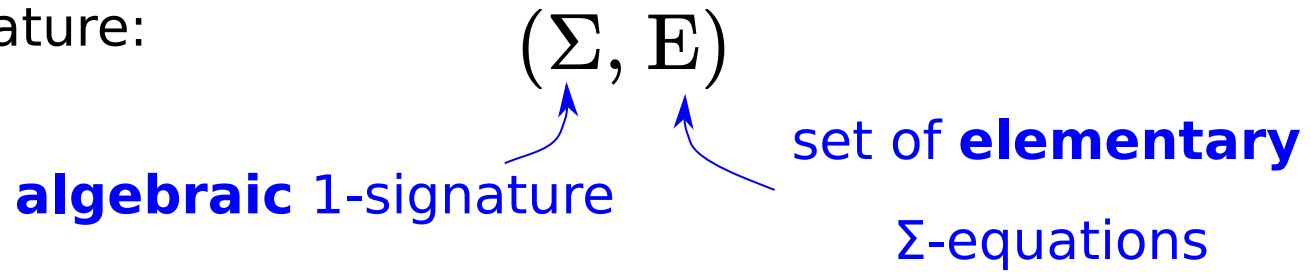
set of Σ -equations (points to E)

model of a 2-signature (Σ, E) :

- a model R of Σ
- s.t. $\forall (A \rightrightarrows B) \in E$, the two morphisms $A(R) \rightrightarrows B(R)$ are equal

Initial semantics for algebraic 2-signatures

Algebraic 2-signature:



Theorem

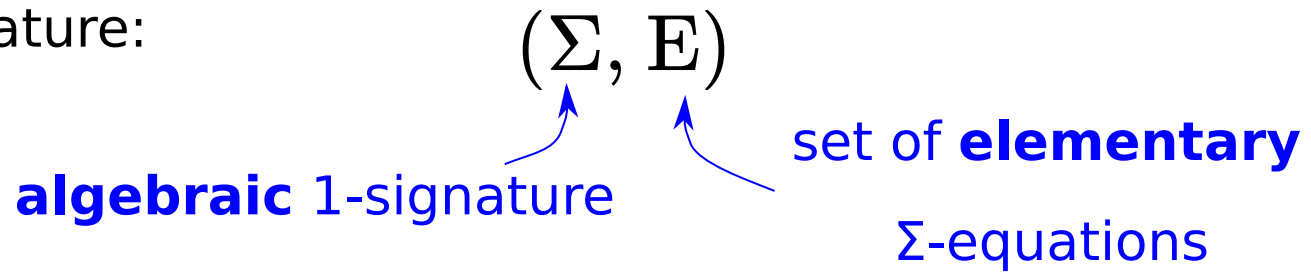
Syntax exists for any algebraic 2-signature.

Main instances of **elementary** Σ -equations $A \Rightarrow B$:

- $A =$ algebraic 1-signature e.g. $A(R) = R \times R$
- $B(R) = R$

Initial semantics for algebraic 2-signatures

Algebraic 2-signature:



Theorem

Syntax exists for any algebraic 2-signature.

Main instances of **elementary** Σ -equations $A \Rightarrow B$:

- $A =$ algebraic 1-signature e.g. $A(R) = R \times R$
- $B(R) = R$

Sketch of the construction of the syntax:

Quotient the initial model R of Σ by the following relation:

$x \sim y$ in $R(X)$ iff for any model S of (Σ, E) , $i(x) = i(y)$

initial Σ -model morphism $i : R \rightarrow S$

Example: λ -calculus modulo $\beta\eta$

The algebraic 2-signature $(\Sigma_{\text{LC}\beta\eta}, E_{\text{LC}\beta\eta})$ of λ -calculus modulo $\beta\eta$:

$$\Sigma_{\text{LC}\beta\eta}(\mathbf{R}) := \Sigma_{\text{LC}}(\mathbf{R}) = (\mathbf{R} \times \mathbf{R}) \amalg \mathbf{R}'$$

model of Σ_{LC} = monad \mathbf{R} with module morphisms:

$$\text{app} : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R} \qquad \text{abs} : \mathbf{R}' \rightarrow \mathbf{R}$$

β -equation: $(\lambda x.t) u = \underbrace{t[x \mapsto u]}_{\sigma_{\mathbf{R}}(t,u)}$

η -equation: $t = \lambda x.(t x)$

$$E_{\text{LC}\beta\eta} = \{ \beta\text{-equation}, \eta\text{-equation} \}$$

Example: λ -calculus modulo $\beta\eta$

The algebraic 2-signature $(\Sigma_{\text{LC}\beta\eta}, E_{\text{LC}\beta\eta})$ of λ -calculus modulo $\beta\eta$:

$$\Sigma_{\text{LC}\beta\eta}(\mathbf{R}) := \Sigma_{\text{LC}}(\mathbf{R}) = (\mathbf{R} \times \mathbf{R}) \amalg \mathbf{R}'$$

model of Σ_{LC} = monad \mathbf{R} with module morphisms:

$$\text{app} : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R} \quad \text{abs} : \mathbf{R}' \rightarrow \mathbf{R}$$

β -equation: $(\lambda x.t) u = t[\underbrace{x \mapsto u}_{\sigma_{\mathbf{R}}(t,u)}]$

η -equation: $t = \lambda x.(t x)$

$$\begin{array}{ccccc}
 & \sigma_{\mathbf{R}} & & & \\
 \mathbf{R}' \times \mathbf{R} & \xrightarrow{\quad} & \mathbf{R} & & \\
 \text{abs} \times \mathbf{R} \searrow & & \nearrow \text{app} & & \\
 & \mathbf{R} \times \mathbf{R} & & &
 \end{array}$$

$$\begin{array}{ccccc}
 & \text{id}_{\mathbf{R}} & & & \\
 \mathbf{R} & \xrightarrow{\quad} & \mathbf{R} & & \\
 \mathbf{R} \downarrow \text{t}_1 & & \nearrow \text{abs} & & \\
 & \mathbf{R}' & & &
 \end{array}$$

$$E_{\text{LC}\beta\eta} = \{ \beta\text{-equation}, \eta\text{-equation} \}$$

Example: fixpoint operator

Definition [AHLM CSL 2018]

A **fixpoint operator** in a monad R is a module morphism $\text{fix}: R' \rightarrow R$ s.t. for any term $t \in R(X \amalg \{\diamond\})$, $\text{fix}(t) = t[\diamond \mapsto \text{fix}(t)]$

Intuition: $\text{fix}(t) := \text{let rec } \diamond = t \text{ in } \diamond$

Algebraic 2-signature $(\Sigma_{\text{fix}}, E_{\text{fix}})$ of a fixpoint operator:

$$\Sigma_{\text{fix}}(R) := R'$$

$$E_{\text{fix}} = \left\{ \begin{array}{ccc} & \xrightarrow{\text{fix}(t)} & \\ R' & & R \\ & \xleftarrow{t[\diamond \mapsto \text{fix}(t)]} & \\ & \text{t} & \end{array} \right\}$$

Proposition [AHLM CSL 2018]

Fixpoint operators in $\text{LC}_{\beta\eta}$ are in one to one correspondance with fixpoint combinators (i.e. λ -terms Y s.t. $t(Yt) = Yt$ for any t).

Combining algebraic 2-signatures

Algebraic 2-signatures can be combined:

fixpoint operator

λ -calculus modulo $\beta\eta$

$(\Sigma_{\text{fix}}, E_{\text{fix}})$

+

$(\Sigma_{\text{LC}\beta\eta}, E_{\text{LC}\beta\eta})$

=

$(\Sigma_{\text{fix}} \amalg \Sigma_{\text{LC}\beta\eta}, E_{\text{fix}} \cup E_{\text{LC}\beta\eta})$

λ -calculus modulo $\beta\eta$ with an explicit fixpoint operator

Example: free commutative monoid

An algebraic 2-signature $(\Sigma_{\text{mon}}, E_{\text{mon}})$ for the free commutative monoid

monad:

$$\Sigma_{\text{mon}}(\mathbf{R}) := 1 \coprod (\mathbf{R} \times \mathbf{R})$$

model of Σ_{mon} = monad \mathbf{R} with module morphisms:

$$0 : 1 \rightarrow \mathbf{R} \quad + : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$$

Example: free commutative monoid

An algebraic 2-signature $(\Sigma_{\text{mon}}, \mathbf{E}_{\text{mon}})$ for the free commutative monoid

monad: $\Sigma_{\text{mon}}(\mathbf{R}) := 1 \amalg (\mathbf{R} \times \mathbf{R})$

model of Σ_{mon} = monad \mathbf{R} with module morphisms:

$$0 : 1 \rightarrow \mathbf{R} \quad + : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$$

4 elementary Σ -equations:

$$\begin{array}{ccc} R & \xrightarrow{0+t} & R \\ & \searrow t & \nearrow \\ & R & \end{array}$$

$$\begin{array}{ccc} R \times R \times R & \xrightarrow{(s+t)+u} & R \\ & \searrow s, t, u & \nearrow \\ & R & \end{array}$$

$$\begin{array}{ccc} R & \xrightarrow{t+0} & R \\ & \searrow t & \nearrow \\ & R & \end{array}$$

$$\begin{array}{ccc} R \times R & \xrightarrow{s+t} & R \\ & \searrow s, t & \nearrow \\ & R & \end{array}$$

Our target: dLC

Syntax of the *differential λ -calculus*:

Differential λ -terms

$$\begin{array}{lcl} s, t & ::= & x \\ & | & \lambda x. t \\ & | & s \ t \\ & | & Ds \cdot t \\ & | & s + t \\ & | & 0 \end{array} \quad \left. \begin{array}{l} \\ \\ \\ \\ \end{array} \right\} \begin{array}{l} \lambda\text{-calculus} \\ \\ \\ \text{free commutative monoid} \end{array}$$

and (bi)linearity of constructors with respect to +:

$$\lambda x. (s + t) = \lambda x. s + \lambda x. t \quad \dots$$

Algebraic 1-signature for dLC

Syntax of the *differential λ -calculus*:

Differential λ -terms

Corresponding 1-signature

$s, t ::= x$

| $\lambda x. t$

| $s \ t$

| $Ds \cdot t$

| $s + t$

| 0

}

$\Sigma_{\text{LC}}(\mathbf{R}) = \mathbf{R}' \coprod (\mathbf{R} \times \mathbf{R})$

$\mathbf{R} \mapsto \mathbf{R} \times \mathbf{R}$

}

$\Sigma_{\text{mon}}(\mathbf{R}) = 1 \coprod (\mathbf{R} \times \mathbf{R})$

Algebraic 1-signature for dLC

Syntax of the *differential λ -calculus*:

Differential λ -terms

Corresponding 1-signature

$s, t ::= x$	
$\lambda x. t$	}
$s t$	
$Ds \cdot t$	
$s + t$	}
0	

$\Sigma_{\text{LC}}(\mathbf{R}) = \mathbf{R}' \coprod (\mathbf{R} \times \mathbf{R})$

$\mathbf{R} \mapsto \mathbf{R} \times \mathbf{R}$

$\Sigma_{\text{mon}}(\mathbf{R}) = 1 \coprod (\mathbf{R} \times \mathbf{R})$

Resulting algebraic 1-signature:

$$\Sigma_{\text{dLC}}(\mathbf{R}) = \Sigma_{\text{LC}}(\mathbf{R}) \coprod (\mathbf{R} \times \mathbf{R}) \coprod \Sigma_{\text{mon}}(\mathbf{R})$$

Elementary equations for dLC

Commutative monoidal structure:

$$\mathbf{E}_{\text{mon}} \left\{ \begin{array}{l} s + t = t + s \\ s + (t + u) = (s + t) + u \\ 0 + t = t \\ t + 0 = t \end{array} \right. \quad \begin{array}{l} \mathbf{R} \times \mathbf{R} \Rightarrow \mathbf{R} \\ \mathbf{R} \times \mathbf{R} \times \mathbf{R} \Rightarrow \mathbf{R} \\ \mathbf{R} \Rightarrow \mathbf{R} \\ \mathbf{R} \Rightarrow \mathbf{R} \end{array}$$

Linearity:

$$\begin{array}{ll} \lambda x. (s + t) = \lambda x. s + \lambda x. t & \mathbf{R} \times \mathbf{R} \Rightarrow \mathbf{R} \\ D(s + t) \cdot u = Ds \cdot u + Dt \cdot u & \mathbf{R} \times \mathbf{R} \times \mathbf{R} \Rightarrow \mathbf{R} \\ Ds \cdot (t + u) = Ds \cdot t + Ds \cdot u & \mathbf{R} \times \mathbf{R} \times \mathbf{R} \Rightarrow \mathbf{R} \end{array}$$

...

n-ary fixpoint operator

Reminder: unary fixpoint operator in a monad R

$$\begin{array}{ccc} R(X \coprod \{\diamond\}) & \rightarrow & R(X) \\ t & \mapsto & \bar{t} \end{array} \quad \textbf{s.t.} \quad t[\diamond \mapsto \bar{t}] = \bar{t}$$

Intuition: $\bar{t} := \text{let rec } \diamond = t \text{ in } \diamond$

n-ary fixpoint operator:

$$\forall \textcolor{red}{i} \in \{1, \dots, n\}, \quad \begin{array}{ccc} R(X \coprod \{\diamond_1, \dots, \diamond_n\})^{\textcolor{blue}{n}} & \rightarrow & R(X) \\ t_1, \dots, t_n & \mapsto & \bar{t}_{\textcolor{red}{i}} \end{array} \quad \textbf{s.t.} \quad \forall i, t_i \left[\begin{array}{c} \diamond_1 \mapsto \bar{t}_1 \\ \dots \\ \diamond_n \mapsto \bar{t}_n \end{array} \right] = \bar{t}_i$$

Intuition:

$\bar{t}_{\textcolor{red}{i}} := \text{let rec } \diamond_1 = t_1 \text{ and } \dots \text{ and } \diamond_n = t_n \text{ in } \diamond_{\textcolor{red}{i}}$

n-ary fixpoint operator

n-ary fixpoint operator:

$$\begin{array}{ll} \forall \textcolor{red}{i} \in \{1, \dots, n\}, & \\ R(X \coprod \{\diamond_1, \dots, \diamond_n\})^{\textcolor{blue}{n}} & \rightarrow R(X) \\ t_1, \dots, t_n & \mapsto \overline{t_{\textcolor{red}{i}}} \end{array} \quad \textbf{s.t.} \quad \forall i, \quad t_i \left[\begin{array}{c} \diamond_1 \mapsto \overline{t_1} \\ \dots \\ \diamond_n \mapsto \overline{t_n} \end{array} \right] = \overline{t_i}$$

Algebraic 1-signature:

$$\Sigma_n(R) = \prod_{i=1}^n (R^{\overbrace{'\dots'}^{\textcolor{blue}{n}}})^n$$

n elementary equations $(R^{\overbrace{'\dots'}^{\textcolor{blue}{n}}})^n \Rightarrow R$

$$\forall i, \quad t_i \left[\begin{array}{c} \diamond_1 \mapsto \overline{t_1} \\ \dots \\ \diamond_n \mapsto \overline{t_n} \end{array} \right] = \overline{t_i}$$

Fixpoint operators

Syntax with fixpoint operators:

- for each n , a n -ary operator:

`let rec $\diamond_1 = t_1$ and .. and $\diamond_n = t_n$ in \diamond_i`

- compatibility between these operators [AHLM CSL 2018]

Fixpoint operators

Syntax with fixpoint operators:

- for each n , a n -ary operator:

`let rec $\diamond_1 = t_1$ and .. and $\diamond_n = t_n$ in \diamond_i`

- compatibility between these operators [AHLM CSL 2018]

- invariance under **permutation**:

<code>let rec $\diamond_1 = t_1$</code>		<code>let rec $\diamond_1 = t_2[\diamond_1 \leftrightarrow \diamond_2]$</code>
<code>and $\diamond_2 = t_2$</code>	<code>=</code>	<code>and $\diamond_2 = t_1[\diamond_1 \leftrightarrow \diamond_2]$</code>
<code>in \diamond_1</code>		<code>in \diamond_2</code>

Fixpoint operators

Syntax with fixpoint operators:

- for each n , a n -ary operator:

`let rec $\diamond_1 = t_1$ and .. and $\diamond_n = t_n$ in \diamond_i`

- compatibility between these operators [AHLM CSL 2018]

- invariance under **permutation**:

<code>let rec $\diamond_1 = t_1$</code>		<code>let rec $\diamond_1 = t_2[\diamond_1 \leftrightarrow \diamond_2]$</code>
<code>and $\diamond_2 = t_2$</code>	<code>=</code>	<code>and $\diamond_2 = t_1[\diamond_1 \leftrightarrow \diamond_2]$</code>
<code>in \diamond_1</code>		<code>in \diamond_2</code>

- invariance under **repetition**:

<code>let rec $\diamond_1 = t$</code>		<code>let rec $\diamond_1 = t[\diamond_2 \mapsto \diamond_1]$</code>
<code>and $\diamond_2 = t$</code>	<code>=</code>	<code>in \diamond_1</code>
<code>in \diamond_1</code>		

Fixpoint operators

Syntax with fixpoint operators:

- for each n , a n -ary operator:

`let rec $\diamond_1 = t_1$ and .. and $\diamond_n = t_n$ in \diamond_i`

- compatibility between these operators [AHLM CSL 2018]

In general:

$$\boxed{\begin{array}{l} \text{let rec } \diamond_1 = t_1 [\diamond_i \mapsto \diamond_{u(i)}] \\ \quad \dots \\ \text{and } \diamond_q = t_q [\diamond_i \mapsto \diamond_{u(i)}] \\ \text{in } \diamond_{u(j)} \end{array}} = \boxed{\begin{array}{l} \text{let rec } \diamond_1 = t_{u(1)} \\ \quad \dots \\ \text{and } \diamond_p = t_{u(p)} \\ \text{in } \diamond_j \end{array}}$$

where $u : \{1, \dots, p\} \rightarrow \{1, \dots, q\}$

$$t_1, \dots, t_q \in R(X \amalg \{\diamond_1, \dots, \diamond_p\})$$

Fixpoint operators

Syntax with fixpoint operators:

- for each n , a n -ary operator:

`let rec $\diamond_1 = t_1$ and .. and $\diamond_n = t_n$ in \diamond_i`

- compatibility between these operators [AHLM CSL 2018]

In general:

$$\boxed{\begin{array}{l} \text{let rec } \diamond_1 = t_1 [\diamond_i \mapsto \diamond_{u(i)}] \\ \quad \dots \\ \text{and } \diamond_q = t_q [\diamond_i \mapsto \diamond_{u(i)}] \\ \text{in } \diamond_{u(j)} \end{array}} = \boxed{\begin{array}{l} \text{let rec } \diamond_1 = t_{u(1)} \\ \quad \dots \\ \text{and } \diamond_p = t_{u(p)} \\ \text{in } \diamond_j \end{array}}$$

where $u : \{1, \dots, p\} \rightarrow \{1, \dots, q\}$

$$t_1, \dots, t_q \in R(X \amalg \{\diamond_1, \dots, \diamond_p\})$$

\Rightarrow Expressible as elementary equations $(R', \dots)^q \Rightarrow R$.

Table of contents

1. 1-Signatures and models based on monads and modules


2. Equations

3. Recursion

Principle of recursion

Recursion on the syntax \simeq Initiality in the category of models

Recipe for constructing "by recursion" a monad morphism:

$f : R \rightarrow S$

initial model of a 2-signature (Σ, E)

Principle of recursion

Recursion on the syntax \approx Initiality in the category of models

Recipe for constructing "by recursion" a monad morphism:

$$f : R \rightarrow S$$

initial model of a 2-signature (Σ, E)



1. Give a module morphism $s : \Sigma(S) \rightarrow S$

Principle of recursion

Recursion on the syntax \approx Initiality in the category of models

Recipe for constructing "by recursion" a monad morphism:

$$f : R \rightarrow S$$

initial model of a 2-signature (Σ, E)



1. Give a module morphism $s : \Sigma(S) \rightarrow S$
 \Rightarrow induces a Σ -model (S, s)

Principle of recursion

Recursion on the syntax \simeq Initiality in the category of models

Recipe for constructing "by recursion" a monad morphism:

$$f : R \rightarrow S$$

initial model of a 2-signature (Σ, E)



1. Give a module morphism $s : \Sigma(S) \rightarrow S$
 \Rightarrow induces a Σ -model (S, s)
2. Show that all the equations in E are satisfied for this model

Principle of recursion

Recursion on the syntax \simeq Initiality in the category of models

Recipe for constructing "by recursion" a monad morphism:

$$f : R \rightarrow S$$

initial model of a 2-signature (Σ, E)




1. Give a module morphism $s : \Sigma(S) \rightarrow S$
 \Rightarrow induces a Σ -model (S, s)
2. Show that all the equations in E are satisfied for this model
 \Rightarrow induces a model of (Σ, E)

Principle of recursion

Recursion on the syntax \approx Initiality in the category of models

Recipe for constructing "by recursion" a monad morphism:

$f : R \rightarrow S$

initial model of a 2-signature (Σ, E)

1. Give a module morphism $s : \Sigma(S) \rightarrow S$
 \Rightarrow induces a Σ -model (S, s)
2. Show that all the equations in E are satisfied for this model
 \Rightarrow induces a model of (Σ, E)

Initiality of $R \Rightarrow$ model morphism $R \rightarrow S \Rightarrow$ monad morphism $R \rightarrow S$

Example: Computing the set of free variables

LC = initial model of (Σ_{LC}, \emptyset)

$$\Sigma_{LC}(R) = (R \times R) \amalg R'$$

\mathcal{P} = power set monad

Definition of a (monad) morphism $fv : LC \rightarrow \mathcal{P}$ **s.t.**

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

$$fv(\text{abs}(t)) = fv(t) \setminus \{\diamond\}$$

Example: Computing the set of free variables

LC = initial model of (Σ_{LC}, \emptyset)

$$\Sigma_{LC}(R) = (R \times R) \amalg R'$$

\mathcal{P} = power set monad

Definition of a (monad) morphism $fv : LC \rightarrow \mathcal{P}$ **s.t.**

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

$$fv(\text{abs}(t)) = fv(t) \setminus \{\diamond\}$$

\Rightarrow make \mathcal{P} a model of Σ_{LC} :

$$\cup : \mathcal{P} \times \mathcal{P} \rightarrow \mathcal{P}$$

$$-\setminus\{\diamond\} : \mathcal{P}' \rightarrow \mathcal{P}$$

Example: Computing the set of free variables

LC = initial model of (Σ_{LC}, \emptyset)

$$\Sigma_{LC}(R) = (R \times R) \amalg R'$$

\mathcal{P} = power set monad

Definition of a (monad) morphism $fv : LC \rightarrow \mathcal{P}$ **s.t.**

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

$$fv(\text{abs}(t)) = fv(t) \setminus \{\diamond\}$$

\Rightarrow make \mathcal{P} a model of Σ_{LC} :

$$\cup : \mathcal{P} \times \mathcal{P} \rightarrow \mathcal{P}$$

$$-\setminus\{\diamond\} : \mathcal{P}' \rightarrow \mathcal{P}$$

Initiality of $LC \Rightarrow$ $fv : LC \rightarrow \mathcal{P}$ satisfying the above equations (as a model morphism).

Example: Translating λ -calculus with fixpoint

$LC_{\beta\eta\text{fix}}$ = initial model of $(\Sigma_{LC\beta\eta}, E_{LC\beta\eta}) + (\Sigma_{\text{fix}}, E_{\text{fix}})$

λ -calculus modulo $\beta\eta$ with a fixpoint operator $\text{fix} : LC_{\beta\eta\text{fix}}' \rightarrow LC_{\beta\eta\text{fix}}$

$LC_{\beta\eta}$ = initial model of $(\Sigma_{LC\beta\eta}, E_{LC\beta\eta})$

λ -calculus modulo $\beta\eta$

monad morphism

Definition of a translation $f : LC_{\beta\eta\text{fix}} \rightarrow LC_{\beta\eta}$ **s.t.**

$$f(u) = "u[\text{fix}(t) \mapsto \text{app}(Y, \text{abs}(t))]"$$

a chosen fixpoint combinator

Example: Translating λ -calculus with fixpoint

$LC_{\beta\eta\text{fix}}$ = initial model of $(\Sigma_{LC\beta\eta}, E_{LC\beta\eta}) + (\Sigma_{\text{fix}}, E_{\text{fix}})$

λ -calculus modulo $\beta\eta$ with a fixpoint operator $\text{fix} : LC_{\beta\eta\text{fix}}' \rightarrow LC_{\beta\eta\text{fix}}$

$LC_{\beta\eta}$ = initial model of $(\Sigma_{LC\beta\eta}, E_{LC\beta\eta})$

λ -calculus modulo $\beta\eta$

monad morphism

Definition of a translation $f : LC_{\beta\eta\text{fix}} \rightarrow LC_{\beta\eta}$ **s.t.**

$$f(u) = "u[\text{fix}(t) \mapsto \text{app}(\text{Y}, \text{abs}(t))]"$$

a chosen fixpoint combinator

\Rightarrow make $LC_{\beta\eta}$ a model of $(\Sigma_{LC\beta\eta}, E_{LC\beta\eta}) + (\Sigma_{\text{fix}}, E_{\text{fix}})$:

app, abs

$$\hat{Y} : LC_{\beta\eta}' \rightarrow LC_{\beta\eta}$$

$$t \mapsto \text{app}(\text{Y}, \text{abs}(t))$$

Example: Translating λ -calculus with fixpoint

$LC_{\beta\eta\text{fix}}$ = initial model of $(\Sigma_{LC\beta\eta}, E_{LC\beta\eta}) + (\Sigma_{\text{fix}}, E_{\text{fix}})$

λ -calculus modulo $\beta\eta$ with a fixpoint operator $\text{fix} : LC_{\beta\eta\text{fix}}' \rightarrow LC_{\beta\eta\text{fix}}$

$LC_{\beta\eta}$ = initial model of $(\Sigma_{LC\beta\eta}, E_{LC\beta\eta})$

λ -calculus modulo $\beta\eta$

monad morphism

Definition of a translation $f : LC_{\beta\eta\text{fix}} \rightarrow LC_{\beta\eta}$ **s.t.**

$$f(u) = "u[\text{fix}(t) \mapsto \text{app}(\text{Y}, \text{abs}(t))]"$$

a chosen fixpoint combinator

\Rightarrow make $LC_{\beta\eta}$ a model of $(\Sigma_{LC\beta\eta}, E_{LC\beta\eta}) + (\Sigma_{\text{fix}}, E_{\text{fix}})$:

app, abs

$$\hat{Y} : LC_{\beta\eta}' \rightarrow LC_{\beta\eta}$$

$$t \mapsto \text{app}(\text{Y}, \text{abs}(t))$$

Initiality of $LC_{\beta\eta\text{fix}} \Rightarrow f : LC_{\beta\eta\text{fix}} \rightarrow LC_{\beta\eta}$

Example: Computing the size of a term

LC = initial model of (Σ_{LC}, \emptyset)

$$\Sigma_{LC}(R) = (R \times R) \coprod R'$$

Definition of a (monad) morphism $s : LC \rightarrow \mathbb{N}$ s.t.

$$s(\text{app}(t, u)) = 1 + s(t) + s(u)$$

$$s(\text{abs}(t)) = 1 + s(t)$$

Example: Computing the size of a term

LC = initial model of (Σ_{LC}, \emptyset)

$$\Sigma_{LC}(R) = (R \times R) \amalg R'$$

Definition of a ~~(monad)~~ morphism $s : LC \rightarrow \mathbb{N}$ s.t.

$$s(\text{app}(t, u)) = 1 + s(t) + s(u)$$

$$s(\text{abs}(t)) = 1 + s(t)$$



\mathbb{N} is not a monad !

Example: Computing the size of a term

LC = initial model of (Σ_{LC}, \emptyset)

$$\Sigma_{LC}(R) = (R \times R) \amalg R'$$

Definition of a ~~(monad)~~ morphism $s : LC \rightarrow \mathbb{N}$ s.t.

$$s(\text{app}(t, u)) = 1 + s(t) + s(u)$$

$$s(\text{abs}(t)) = 1 + s(t)$$



\mathbb{N} is not a monad !

Solution [CSL AHLM 2018]:

1. define $f : LC \rightarrow \mathbf{C}$ by recursion

2. deduce $s : LC \rightarrow \mathbb{N}$

continuation monad $\mathbf{C}(X) = \mathbb{N}^{\mathbb{N}^X}$

Example: Computing the size of a term

LC = initial model of (Σ_{LC}, \emptyset)

$$\Sigma_{LC}(R) = (R \times R) \amalg R'$$

Definition of a ~~(monad)~~ morphism $s : LC \rightarrow \mathbb{N}$ s.t.

$$s(\text{app}(t, u)) = 1 + s(t) + s(u)$$

$$s(\text{abs}(t)) = 1 + s(t)$$



\mathbb{N} is not a monad !

Solution [CSL AHLM 2018]:

1. define $f : LC \rightarrow \mathbf{C}$ by recursion

2. deduce $s : LC \rightarrow \mathbb{N}$

continuation monad $\mathbf{C}(X) = \mathbb{N}^{(\mathbb{N}^X)}$

assigns an arbitrary size to each variable

Intuition: uncurrying $f_X : LC(X) \rightarrow \mathbb{N}^{(\mathbb{N}^X)}$ yields $g : LC(X) \times \mathbb{N}^X \rightarrow \mathbb{N}$

Example: Computing the size of a term

LC = initial model of (Σ_{LC}, \emptyset)

$$\Sigma_{LC}(R) = (R \times R) \amalg R'$$

Definition of a ~~(monad)~~ morphism $s : LC \rightarrow \mathbb{N}$ s.t.

$$s(\text{app}(t, u)) = 1 + s(t) + s(u)$$

$$s(\text{abs}(t)) = 1 + s(t)$$



\mathbb{N} is not a monad !

Solution [CSL AHLM 2018]:

1. define $f : LC \rightarrow \mathbf{C}$ by recursion

2. deduce $s : LC \rightarrow \mathbb{N}$

continuation monad $\mathbf{C}(X) = \mathbb{N}^{(\mathbb{N}^X)}$

assigns an arbitrary size to each variable

Intuition: uncurrying $f_X : LC(X) \rightarrow \mathbb{N}^{(\mathbb{N}^X)}$ yields $g : LC(X) \times \mathbb{N}^X \rightarrow \mathbb{N}$

$$s(t) = g(t, (x \mapsto 0))$$

variables are of size 0

Conclusion

Summary of the talk:

- presented a notion of 1-signature and models
- defined a 2-signature as a 1-signature and a set of equations
- identified a class of 2-signatures that generate a syntax

The main theorem has been formalized in Coq using the UniMath library.

Future work:

- add the notion of reductions;
- extend our work to simply typed syntaxes.

Conclusion

Summary of the talk:

- presented a notion of 1-signature and models
- defined a 2-signature as a 1-signature and a set of equations
- identified a class of 2-signatures that generate a syntax

The main theorem has been formalized in Coq using the UniMath library.

Future work:

- add the notion of reductions;
- extend our work to simply typed syntaxes.

Thank you!