

# **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

**Syntax**

Model

Operation/Construction

Arity/Signature

**This talk:** give a mathematical account of this area

# Motivation: dLC

dLC = ***differentiable  $\lambda$ -calculus*** [Ehrard-Regnier 2003].

- Syntax: not straightforward (equations involved).  
e.g.  $s+t = t+s$
- Later articles: alternative presentations of the syntax (+/- verbose).
- No general *scheme* seems to be well-established (except BNF)

## **Our work:**

- a mathematical theory of presentations of monads,
- induces a scheme for presenting syntaxes.

# Motivation: dLC

dLC = ***differentiable  $\lambda$ -calculus*** [Ehrard-Regnier 2003].

- Syntax: not straightforward (equations involved).  
e.g.  $s+t = t+s$
- Later articles: alternative presentations of the syntax (+/- verbose).
- No general *scheme* seems to be well-established (except BNF)

Next slides: 3 variants of the dLC syntax:

## **Our work:**

- a mathematical theory of presentations of monads,
- induces a scheme for presenting syntaxes.

# Syntax of dLC: version 1/3

A **syntax** for the ***differentiable  $\lambda$ -calculus*** by ***mutual induction***:

[Categorical Models for Simply Typed Resource Calculi]

***Simple terms:***

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

***Differential  $\lambda$ -terms:***


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

# Syntax of dLC: version 1/3

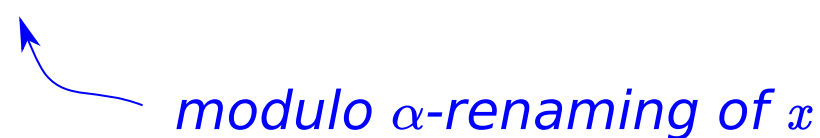
A **syntax** for the ***differentiable  $\lambda$ -calculus*** by ***mutual induction***:

[Categorical Models for Simply Typed Resource Calculi]

**Simple terms:**

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


**Differential  $\lambda$ -terms:**

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



*modulo  $\alpha$ -renaming of  $x$*

# Syntax of dLC: version 1/3



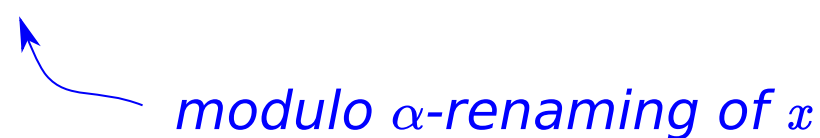
A **syntax** for the ***differentiable  $\lambda$ -calculus*** by ***mutual induction***:

[Categorical Models for Simply Typed Resource Calculi]

**Simple terms:**

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


**Differential  $\lambda$ -terms:**

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


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

# Syntax of dLC: version 1/3

A **syntax** for the ***differentiable  $\lambda$ -calculus*** by ***mutual induction***:

[Categorical Models for Simply Typed Resource Calculi]

**Simple terms:**

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

*variable* (arrow from text to  $x$ )

**Differential  $\lambda$ -terms:**

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

*modulo  $\alpha$ -renaming of  $x$*  (arrow from text to  $\lambda x. s$ )

*neutral element for  $+$*  (arrow from text to  $0$ )

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

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

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

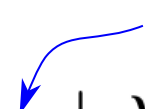


# Syntax of dLC: version 1/3



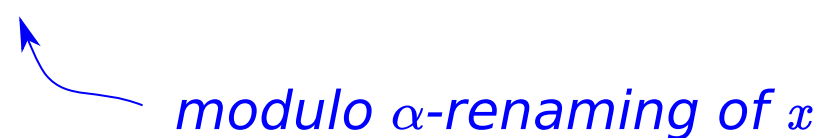
A **syntax** for the ***differentiable  $\lambda$ -calculus*** by ***mutual induction***:

[Categorical Models for Simply Typed Resource Calculi]

**Simple terms:**

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


**Differential  $\lambda$ -terms:**

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


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

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

But which ones are allowed ? What is the limit ?

# Syntax of dLC: version 2/3

**Which operations/equations are allowed to specify a syntax ?**

**A stand-alone presentation of simple terms:**

*Simple terms:*

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

*Differential  $\lambda$ -terms:*

$$T \in \Lambda^d = \text{FreeCommutativeMonoid}(\Lambda^s)$$

# Syntax of dLC: version 2/3

**Which operations/equations are allowed to specify a syntax ?**

**A stand-alone presentation of simple terms:**

*Simple terms:*

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

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



*Differential  $\lambda$ -terms:*

$$T \in \Lambda^d = \text{FreeCommutativeMonoid}(\Lambda^s)$$

# Syntax of dLC: version 3/3

**Which operations/equations are allowed to specify a syntax ?**

**A stand-alone presentation of differential  $\lambda$ -terms:**

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

*Differential  $\lambda$ -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*

[Categorical Models for Simply  
Typed Resource Calculi]

$$\lambda x. \sum_i t_i = \sum_i \lambda x. t_i$$

$$(\sum_i t_i) u = \sum_i t_i u$$

$$D(\sum_i t_i) \cdot (\sum_j u_j) = \sum_i \sum_j D t_i \cdot u_j$$

# 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

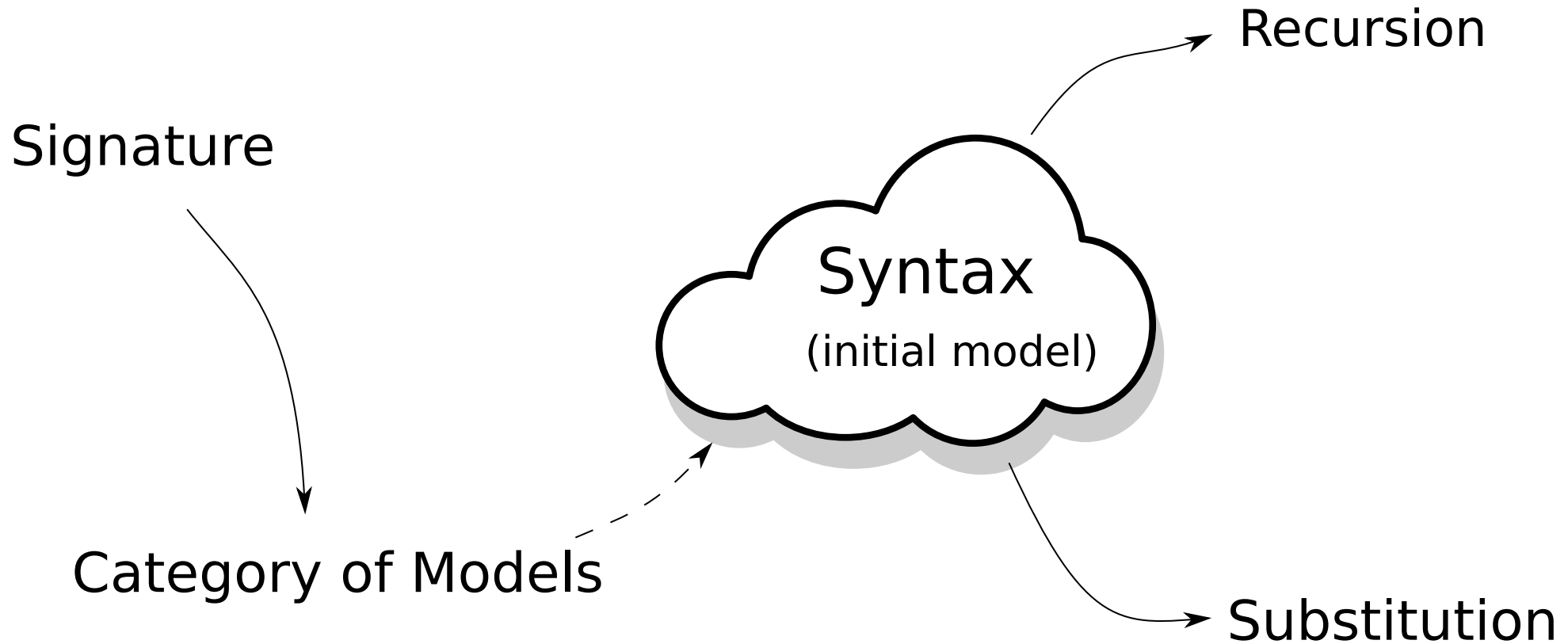
How can we compare these different versions ?

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

# Overview

**Topic:** specification of untyped syntaxes (e.g. differential  $\lambda$ -calculus).

## Our work:

1. general notion of **1-signature** based on **monads** and **modules**.
  - *Caveat:* Not all of them do **generate a syntax**
  - special case: classical **algebraic 1-signatures** generate a syntax
2. notion of **2-signature**: a pair of a 1-signature and a set of equations.
  - special case: **algebraic 2-signatures** generate a syntax



# Related work of Fiore-Hur 2010

**[Fiore-Hur 2010]:** presentations of simply typed languages with

- generating *binding* operations (e.g.  $\lambda$ -abstraction)
- equations among them.

**Our work:** a variant of their approach

- for the untyped setting,
- focus on *monads* and *modules* over them

# Table of contents

- 1. Review: Binding signatures and their models**
2. 1-Signatures and models based on monads and modules
3. Equations
4. Recursion

# Table of contents

## **1. Review: Binding signatures and their models**

- Categorical formulation of term languages
- Initial semantics for binding signatures

## 2. 1-Signatures and models based on monads and modules

## 3. Equations

## 4. Recursion

# Categorical formulation of a term language

**Example:** differential  $\lambda$ -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:**

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

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

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

# Categorical formulation of a term language

**Example:** differential  $\lambda$ -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:**

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

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

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

**Free variable renaming:**

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

# Categorical formulation of a term language

**Example:** differential  $\lambda$ -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:**

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

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

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

**Free variable renaming:**

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

$\Rightarrow$  **dLC is an endofunctor on Set**

# Categorical formulation of a term language

Operations as **natural** transformations:  commute with variable renaming

$$+ : dLC \times dLC \rightarrow dLC$$

$$0 : 1 \rightarrow dLC$$

...

Variables as a natural transformation:

$$\text{var} : \text{Id}_{\text{Set}} \rightarrow dLC$$

# Categorical formulation of a term language

commute with variable renaming

**Operations as natural transformations:**

$$+ : dLC \times dLC \rightrightarrows dLC$$

$$0 : 1 \rightrightarrows dLC$$

...

**Variables as a natural transformation:**

$$\text{var} : \text{Id}_{\text{Set}} \rightrightarrows dLC$$

This gives a notion of model for the language  $(+ , 0)$ :

**model** = endofunctor  $R$  with natural transformations:

$$+ : R \times R \rightrightarrows R$$

$$0 : 1 \rightrightarrows R$$

$$\text{var} : \text{Id} \rightrightarrows R$$

$$\text{or} \quad (R \times R) \coprod 1 \coprod \text{Id} \rightrightarrows R$$



# Categorical formulation of a term language

commute with variable renaming

**Operations as natural transformations:**

$$+ : dLC \times dLC \rightrightarrows dLC$$

$$0 : 1 \rightrightarrows dLC$$

...

**Variables as a natural transformation:**

$$\text{var} : \text{Id}_{\text{Set}} \rightrightarrows dLC$$

This gives a notion of model for the language  $(+ , 0)$ :

**model** = endofunctor  $R$  with natural transformations:

$$+ : R \times R \rightrightarrows R$$

$$0 : 1 \rightrightarrows R$$

$$\text{var} : \text{Id} \rightrightarrows R$$

$$\text{or} \quad (R \times R) \coprod 1 \coprod \text{Id} \rightrightarrows R$$

Next slides: **generalize this pattern to other languages**

# Binding Signatures

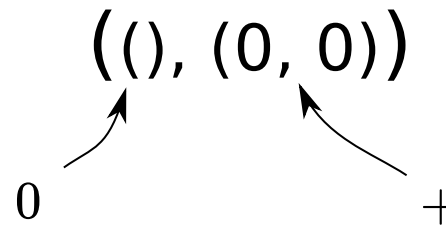
## Definition

**Binding signature** = a family of lists of natural numbers.

Each list specifies one operation in the syntax:

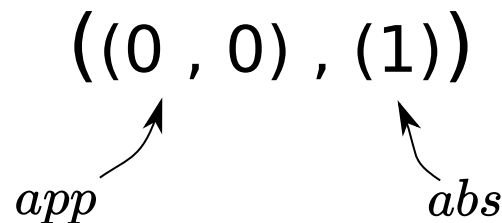
- length of the list = number of arguments of the operation
- natural number in the list = number of bound variables in the corresponding argument

**Syntax with 0, +:**



$0, 0+x, x+y$   
 $(0+x)+y, 0+(x+y),$   
...

**Lambda calculus:**



# Initial semantics for binding signatures

**model** of  $(0, +)$  = endofunctor  $R$  with a natural transformation:

$$[+, 0, \text{var}] : (R \times R) \amalg 1 \amalg \text{Id} \rightarrowtail R$$

**morphism** = natural transformation commuting with  $0$ ,  $+$  and  $\text{var}$ .

Similarly, any binding signature gives rise to a category of models.

Well-established theorem

The initial model of a binding signature  $\Sigma$  always exists.

**Question:** Does this initial model come with a **well-behaved substitution**?

**Answer:** Yes: see e.g. [Fiore, Plotkin, Turi 1999], [Ghani & Uustalu 2003]

# Initial semantics for binding signatures

**model** of  $(0, +)$  = endofunctor  $R$  with a natural transformation:

$$[+, 0, \text{var}] : (R \times R) \amalg 1 \amalg \text{Id} \rightarrowtail R$$

**morphism** = natural transformation commuting with  $0$ ,  $+$  and  $\text{var}$ .

Similarly, any binding signature gives rise to a category of models.

Well-established theorem

The initial model of a binding signature  $\Sigma$  always exists.

Cf next section

**Question:** Does this initial model come with a **well-behaved**  
**substitution**?

**Answer:** Yes: see e.g. [Fiore, Plotkin, Turi 1999], [Ghani & Uustalu 2003]

# Initial semantics for binding signatures

**model** of  $(0, +)$  = endofunctor  $R$  with a natural transformation:

$$[+, 0, \text{var}] : (R \times R) \amalg 1 \amalg \text{Id} \rightarrowtail R$$

**morphism** = natural transformation commuting with  $0$ ,  $+$  and  $\text{var}$ .

Similarly, any binding signature gives rise to a category of models.

Well-established theorem

The initial model of a binding signature  $\Sigma$  always exists.

Cf next section

**Question:** Does this initial model come with a **well-behaved** substitution?

**Answer:** Yes: see e.g. [Fiore, Plotkin, Turi 1999], [Ghani & Uustalu 2003]

Initiality still holds in the subcategory of models with a substitution.

# Table of contents

1. Review: Binding signatures and their models

## **2. 1-Signatures and models based on monads and modules**

- Our take on substitution
- Our take on 1-signatures, models and syntax
- Our take on binding 1-signatures

3. Equations

4. Recursion

# The Big Picture of 1-signatures and models

Binding signatures  $\hookrightarrow$  Our 1-signatures

A **1-signature**  $\Sigma$  = functorial assignment:

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

**Example:**  $(0, +)$

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

A **model of  $\Sigma$**  is a pair:

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

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

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

monad := endofunctor with substitution


module over a monad := endofunctor with substitution

module morphism := natural transformation preserving substitution

# The Big Picture of 1-signatures and models

Binding signatures  $\hookrightarrow$  Our 1-signatures

A **1-signature**  $\Sigma$  = functorial assignment:

  $R \mapsto \Sigma(R)$   
monad

**Example:**  $(0, +)$

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

A **model of**  $\Sigma$  is a pair:

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

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

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

monad := endofunctor with substitution

module over a monad := endofunctor with substitution

module morphism := natural transformation preserving substitution



# The Big Picture of 1-signatures and models

Binding signatures  $\hookrightarrow$  Our 1-signatures

A **1-signature**  $\Sigma$  = functorial assignment:

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

monad  $\nearrow$   $\nwarrow$  module over  $R$

A **model of**  $\Sigma$  is a pair:

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

**Example:**  $(0, +)$

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

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

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

monad := endofunctor with substitution

module over a monad := endofunctor with substitution

module morphism := natural transformation preserving substitution

# The Big Picture of 1-signatures and models

Binding signatures  $\hookrightarrow$  Our 1-signatures

A **1-signature**  $\Sigma$  = functorial assignment:

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

monad  $\nearrow$   $\nwarrow$  module over  $R$

A **model of**  $\Sigma$  is a pair:

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

monad  $\nearrow$

**Example:**  $(0, +)$

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

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

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

monad := endofunctor with substitution

module over a monad := endofunctor with substitution

module morphism := natural transformation preserving substitution

# The Big Picture of 1-signatures and models

Binding signatures  $\hookrightarrow$  Our 1-signatures

A **1-signature**  $\Sigma$  = functorial assignment:

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

monad  $\nearrow$   $\nwarrow$  module over  $R$

A **model of**  $\Sigma$  is a pair:

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

monad  $\nearrow$   $\nwarrow$  module morphism

**Example:**  $(0, +)$

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

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

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

monad := endofunctor with substitution

module over a monad := endofunctor with substitution

module morphism := natural transformation preserving substitution

# Substitution and monads

## Reminder:

- $dLC(X) = \{ \text{differential } \lambda\text{-terms taking free variables in } X \}$
- Variables as a natural transformation  $\text{var} : \text{Id}_{\text{Set}} \rightarrow dLC$
- **Variable renaming** by functoriality:

$$dLC(f)(t) = t[ x \mapsto f(x) ] \quad \text{where } f : X \rightarrow Y \text{ is a renaming}$$

# Substitution and monads

## Reminder:

- $dLC(X) = \{ \text{differential } \lambda\text{-terms taking free variables in } X \}$
- Variables as a natural transformation  $\text{var} : \text{Id}_{\text{Set}} \rightarrow dLC$
- **Variable renaming** by functoriality:

$$dLC(f)(t) = t[ \underbrace{x \mapsto f(x)} ] \quad \text{where } f : X \rightarrow Y \text{ is a renaming}$$

still makes sense if  $f(x)$  is an arbitrary term (rather than just a variable)

# Substitution and monads

## Reminder:

- $dLC(X) = \{ \text{differential } \lambda\text{-terms taking free variables in } X \}$
- Variables as a natural transformation  $\text{var} : \text{Id}_{\text{Set}} \rightarrow dLC$
- **Variable renaming** by functoriality:

$$dLC(f)(t) = t[\underbrace{x \mapsto f(x)}] \quad \text{where } f : X \rightarrow Y \text{ is a renaming}$$

still makes sense if  $f(x)$  is an arbitrary term (rather than just a variable)

**Variable renaming** = special case of **substitution**:

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

# Substitution and monads

## Reminder:

- $dLC(X) = \{ \text{differential } \lambda\text{-terms taking free variables in } X \}$
- Variables as a natural transformation  $\text{var} : \text{Id}_{\text{Set}} \rightarrow dLC$
- **Variable renaming** by functoriality:

$$dLC(f)(t) = t[\underbrace{x \mapsto f(x)}] \quad \text{where } f : X \rightarrow Y \text{ is a renaming}$$

still makes sense if  $f(x)$  is an arbitrary term (rather than just a variable)

**Variable renaming** = special case of **substitution**:

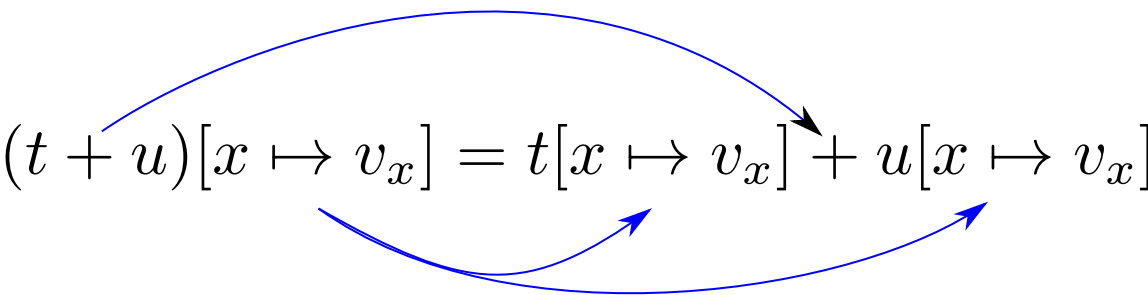
$$\begin{array}{ccc} \text{bind}_f : dLC(X) & \rightarrow & dLC(Y) \\ t & \mapsto & t[x \mapsto f(x)] \end{array} \quad \text{where } f : X \rightarrow dLC(Y)$$

$(dLC, \text{var}, \text{bind}) = \text{a monad.}$

**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 binding signatures

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

- $R$  itself
- $M \times N$  for any modules  $M$  and  $N$  (e.g.,  $R \times R$ )
- $M' = \mathbf{derivative\ of\ a\ module\ } M$ :  $M'(X) = M(X \coprod \{\diamond\})$ .  
used to model an operation binding a variable (Cf next slide).

# Syntactic operations are module morphisms

**module morphism** = maps commuting with substitution.

$$\text{id}_M : M \rightarrow M$$

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

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

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

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

$$\text{abs}_X : \text{dLC}(X + \{\diamond\}) \rightarrow \text{dLC}(X)$$

$$t \mapsto \lambda \diamond . t$$

# The Big Picture again

A **1-signature**  $\Sigma$  = functorial assignment:

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

monad  $\nearrow$   $\nwarrow$  module over  $R$

A **model of  $\Sigma$**  is a pair:

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

monad  $\nearrow$   $\nwarrow$  module morphism

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  $\Sigma$ , 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  $R \mapsto \mathcal{P} \circ R$



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 binding signatures through the embedding:

Binding signatures  $\hookrightarrow$  Our 1-signatures

**Question:** Can we enforce some equations in the syntax ?

For example: commutativity of  $+$  for the differential  $\lambda$ -calculus.

# Table of contents

1. Review: Binding 1-signatures and their models
2. 1-Signatures and models based on monads and modules
- 3. Equations**
4. 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

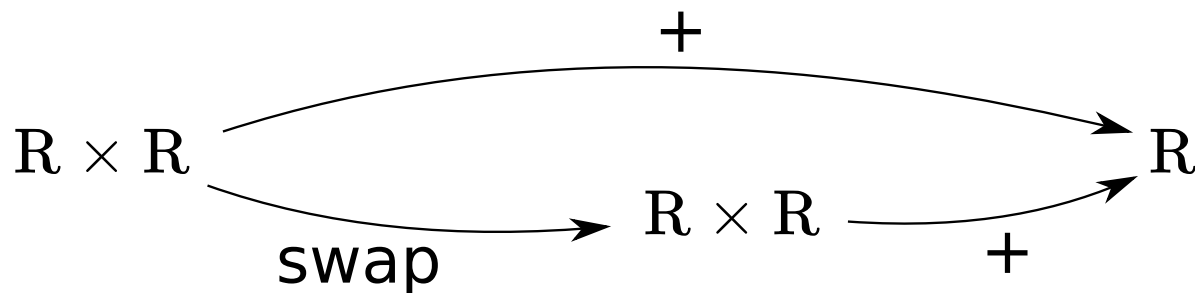
1-Signature:  $R \mapsto R \times R$

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:



# Review: Signatures with equations

- [Fiore-Hur 2010]: inductively defined set of possible equations.
- [AHLM CSL 2018]: "quotients" of algebraic 1-signatures

*Examples:*

- a binary commutative operation
- application of the simple terms of differential  $\lambda$ -calculus (2<sup>nd</sup> variant)

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

# Review: Signatures with equations

- [Fiore-Hur 2010]: inductively defined set of possible equations.

This work: alternative approach where monads and modules are the central notions.

- [AHLM CSL 2018]: "quotients" of algebraic 1-signatures

*Examples:*

- a binary commutative operation
- application of the simple terms of differential  $\lambda$ -calculus (2<sup>nd</sup> variant)

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

# Review: Signatures with equations

- [Fiore-Hur 2010]: inductively defined set of possible equations.

This work: alternative approach where monads and modules are the central notions.

- [AHLM CSL 2018]: "quotients" of algebraic 1-signatures

*Examples:*

- a binary commutative operation
- application of the simple terms of differential  $\lambda$ -calculus (2<sup>nd</sup> variant)

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

This work: more general equations (e.g. associativity of a binary op).

# Equations

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

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

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

model of  $\Sigma$  (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  $\Sigma$ )

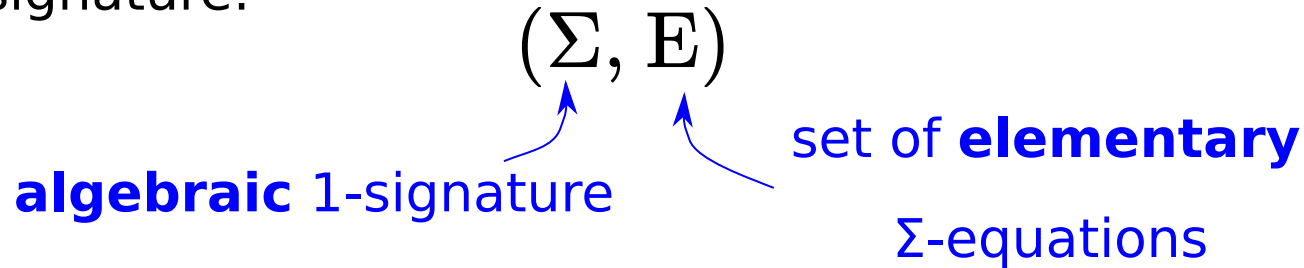
set of  $\Sigma$ -equations (points to  $E$ )

**model of a 2-signature**  $(\Sigma, E)$ :

- a model  $R$  of  $\Sigma$
- 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

Given a 1-signature  $\Sigma$ , a  $\Sigma$ -equation  $A \Rightarrow B$  is **elementary** if:

1.  $A$  "preserves pointwise epimorphisms"

(e.g., any "algebraic 1-signature", such as  $R \mapsto R \times R$ )

2.  $B$  is of the form  $R \mapsto R' \dots'$  (e.g.  $R \mapsto R$ )

# Example: $\lambda$ -calculus modulo $\beta\eta$

The algebraic 2-signature  $(\Sigma_{\text{LC}\beta\eta}, E_{\text{LC}\beta\eta})$  of  $\lambda$ -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  $\Sigma_{\text{LC}}$**  = monad  $\mathbf{R}$  with module morphisms:

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

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

**$\eta$ -equation:**  $t = \lambda x.(t x)$

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



# Example: $\lambda$ -calculus modulo $\beta\eta$

The algebraic 2-signature  $(\Sigma_{\text{LC}\beta\eta}, E_{\text{LC}\beta\eta})$  of  $\lambda$ -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  $\Sigma_{\text{LC}}$**  = monad  $\mathbf{R}$  with module morphisms:

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

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

**$\eta$ -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  $f : R' \rightarrow R$  s.t. for any term  $t \in R(X \amalg \{ \diamond \})$ ,  $f(t) = t[\diamond \mapsto f(t)]$ ,

i.e. (1)  $R' \xrightarrow{\quad f \quad} R$  commutes.

$$\begin{array}{ccccc}
 R' & \xrightarrow{\quad f \quad} & R \\
 \searrow (\text{id}_{R'}, f) & & \nearrow \sigma_R \\
 & R' \times R &
 \end{array}$$

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

$$\Sigma_{\text{fix}}(R) := R' \qquad E_{\text{fix}} = \{ (1) \}$$

Proposition [AHLM CSL 2018]

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

# Combining algebraic 2-signatures

Algebraic 2-signatures can be combined:

fixpoint operator

$\lambda$ -calculus modulo  $\beta\eta$

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

+

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

=

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

$\lambda$ -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**  $\Sigma_{\text{mon}}$  = 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**  $\Sigma_{\text{mon}}$  = monad  $\mathbf{R}$  with module morphisms:

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

4 elementary  $\Sigma$ -equations:

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

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

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

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

# Our target: dLC

## Syntax of the *differentiable $\lambda$ -calculus*:

*Differential  $\lambda$ -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 *differentiable* $\lambda$ -calculus:

*Differential*  $\lambda$ -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 *differentiable* $\lambda$ -calculus:

*Differential  $\lambda$ -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:

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

...


# Table of contents

1. Review: Binding signatures and their models
2. 1-Signatures and models based on monads and modules
3. Equations
- 4. 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  $(\Sigma, E)$

# 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  $(\Sigma, E)$



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  $(\Sigma, E)$



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



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




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

# 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  $(\Sigma, E)$

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

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  $(\Sigma_{LC}, \emptyset)$

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

$\mathcal{P}$  = 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  $(\Sigma_{\text{LC}}, \emptyset)$

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

$\mathcal{P}$  = power set monad

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

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

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

$\Rightarrow$  make  $\mathcal{P}$  a model of  $\Sigma_{\text{LC}}$ :

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

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

# Example: Computing the set of free variables

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

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

$\mathcal{P}$  = 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  $\Sigma_{LC}$ :

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

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

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

# Example: Translating $\lambda$ -calculus with fixpoint

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

*$\lambda$ -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  $(\Sigma_{LC\beta\eta}, E_{LC\beta\eta})$

*$\lambda$ -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 $\lambda$ -calculus with fixpoint

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

*$\lambda$ -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  $(\Sigma_{LC\beta\eta}, E_{LC\beta\eta})$

*$\lambda$ -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 $\lambda$ -calculus with fixpoint

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

*$\lambda$ -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  $(\Sigma_{LC\beta\eta}, E_{LC\beta\eta})$

*$\lambda$ -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  $(\Sigma_{LC}, \emptyset)$

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

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

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

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

# Example: Computing the size of a term

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

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

**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  $(\Sigma_{LC}, \emptyset)$

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

**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  $(\Sigma_{LC}, \emptyset)$

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

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

affects 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  $(\Sigma_{LC}, \emptyset)$

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

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

affects 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!

# Monads

1.  $dLC : Set \rightarrow Set$
2. A collection of functions  $(var_X : X \rightarrow dLC(X))_X$   
*Variables are expressions*
3. For each function  $u : X \rightarrow dLC(Y)$ , a function  $bind_u : dLC(X) \rightarrow dLC(Y)$   
*Parallel substitution*  
**Notation:**  $bind_u(t) = t[x \mapsto u(x)]$
4. Monadic laws:

$$var(y)[x \mapsto u(x)] = u(y)$$

$$t[x \mapsto var(x)] = t$$

$$t[x \mapsto f(x)][y \mapsto g(y)] = t[x \mapsto f(x)[y \mapsto g(y)]]$$

# Modules VS Monads

## Monad

1.  $R : \text{Set} \rightarrow \text{Set}$
2. A collection of functions  $(\text{var}_X : X \rightarrow R(X))_X$   
*Variables are expressions*
3. For each function  $u : X \rightarrow R(Y)$ , a function  $\text{bind}_u : R(X) \rightarrow R(Y)$   
*Parallel substitution*

**Notation:**  $\text{bind}_u(t) = t[x \mapsto u(x)]^R$

4. Substitution laws:

$$\text{var}(y)[x \mapsto u(x)]^R = u(y)$$

$$t[x \mapsto \text{var}(x)]^R = t$$

$$t[x \mapsto f(x)]^R[y \mapsto g(y)]^R = t[x \mapsto f(x)[y \mapsto g(y)]^R]^R$$