

Higher-order Arities, Signatures and Equations via Modules

Ambroise Lafont

joint work with
Benedikt Ahrens, André Hirschowitz, Marco Maggesi

Work submitted to FSCD 2019

Keywords associated with syntax

Induction/Recursion

Substitution

Model

Syntax

Operation/Construction

Arity/Signature

This talk: give a mathematical account of this area

Motivation: LCD

The ***differentiable λ -calculus*** (LCD) was introduced by [Ehrard-Regnier 2003].

The syntax is not straightforward, as it involves some equations.

There are alternative presentations of the syntax in later articles, more or less verbose.

Motivation: LCD

The ***differentiable λ -calculus*** (LCD) was introduced by [Ehrard-Regnier 2003].

The syntax is not straightforward, as it involves some equations.

There are alternative presentations of the syntax in later articles, more or less verbose.

The next slides give 3 variants of the syntax

Syntax of LCD: version 1/3

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

[Categorical Models for Simply Typed Resource Calculi]

Simple terms:

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

Differential λ -terms:


$$\Lambda^d : \quad S, T, U, V ::= 0 \mid s \mid s + T$$

Syntax of LCD: version 1/3


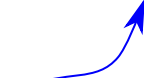

A **syntax** for the ***differentiable λ -calculus*** by ***mutual induction***:

[Categorical Models for Simply Typed Resource Calculi]

Simple terms:

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


Differential λ -terms:

$$\Lambda^d : \quad S, T, U, V ::= 0 \mid s \mid s + T$$


neutral element for +

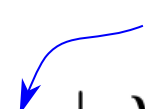
modulo commutativity

Syntax of LCD: version 1/3

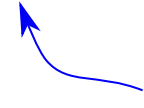
A **syntax** for the ***differentiable λ -calculus*** by ***mutual induction***:

[Categorical Models for Simply Typed Resource Calculi]

Simple terms:

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


Differential λ -terms:

$$\Lambda^d : \quad S, T, U, V ::= 0 \mid s \mid s + T$$


modulo α -renaming of x

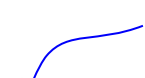
$$\Lambda^d = \text{FreeCommutativeMonoid}(\Lambda^s)$$

Syntax of LCD: version 1/3


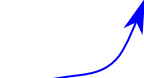

A **syntax** for the ***differentiable λ -calculus*** by ***mutual induction***:

[Categorical Models for Simply Typed Resource Calculi]

Simple terms:

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


Differential λ -terms:

$$\Lambda^d : \quad S, T, U, V ::= 0 \mid s \mid s + T$$


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

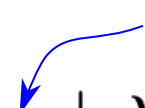
A syntax is specified by operations and **equations**.

Syntax of LCD: version 1/3



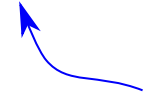
A **syntax** for the ***differentiable λ -calculus*** by ***mutual induction***:

[Categorical Models for Simply Typed Resource Calculi]

Simple terms:

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


Differential λ -terms:

$$\Lambda^d : \quad S, T, U, V ::= 0 \mid s \mid s + T$$


neutral element for +

modulo commutativity

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

A syntax is specified by operations and **equations**.

But which ones are allowed ? What is the limit ?

Syntax of LCD: version 2/3

Which operations/equations are allowed to specify a syntax ?

Can we avoid mutual induction ?

A stand-alone presentation of simple terms:

Simple terms:

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

Differential λ -terms:

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

Syntax of LCD: version 2/3

Which operations/equations are allowed to specify a syntax ?

Can we avoid mutual induction ?

A stand-alone presentation of simple terms:

Simple terms:

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

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



Differential λ -terms:

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

Syntax of LCD: version 3/3

Which operations/equations are allowed to specify a syntax ?

A stand-alone presentation of differential λ -terms:

Allow summands 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

Turn [Categorical Models for
Simply Typed Resource Calculi]'s
abbreviations into equations:

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

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

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

Syntax of LCD: 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 LCD: Conclusion

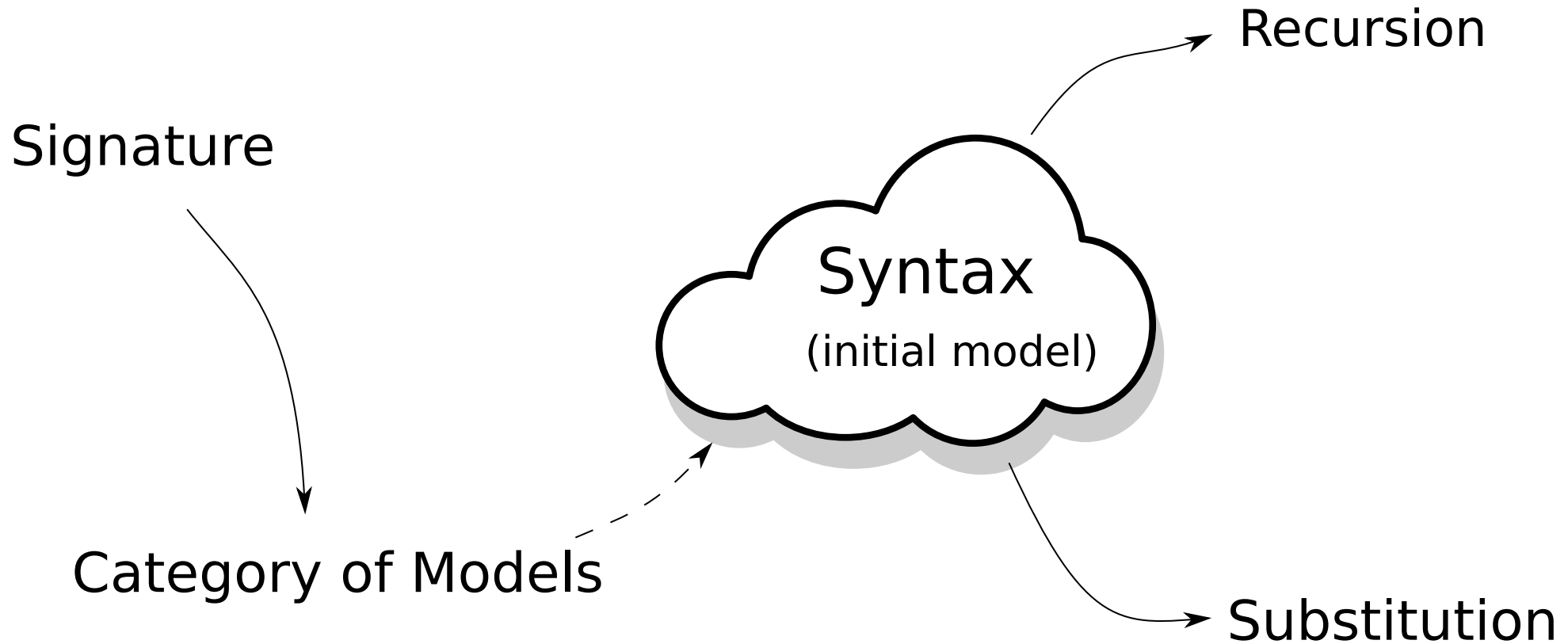
How can we compare these different versions ?

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 and construction of untyped syntaxes with variables and a well-behaved substitution (e.g. differential λ -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

Previous work of Fiore-Hur 2010

[Fiore-Hur 2010]: presentations of simply typed languages by generating *binding* operations (e.g. λ -abstraction) and equations among them.

Our work: for the untyped setting, a variant of their approach where monads and modules over them are the central notions.

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 λ -calculus (last variant)

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

Free variable indexing:

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

$$LCD(\emptyset) = \{0, \lambda x. x, \dots\}$$

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

Categorical formulation of a term language

Example: differential λ -calculus (last variant)

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

Free variable indexing:

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

$$LCD(\emptyset) = \{0, \lambda x. x, \dots\}$$

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

Free variable renaming:

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

Categorical formulation of a term language

Example: differential λ -calculus (last variant)

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

Free variable indexing:

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

$$LCD(\emptyset) = \{0, \lambda x. x, \dots\}$$

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

Free variable renaming:

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

\Rightarrow **LCD is an endofunctor on Set**

Categorical formulation of a term language

Operations as natural transformations:

$$+ : LCD \times LCD \rightrightarrows LCD$$

$$0 : 1 \rightrightarrows LCD$$

...

Variables as a natural transformation:

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

Putting all together:

$$LCD \times LCD + 1 + \text{Id}_{\text{Set}} \rightrightarrows LCD$$

Binding Signatures

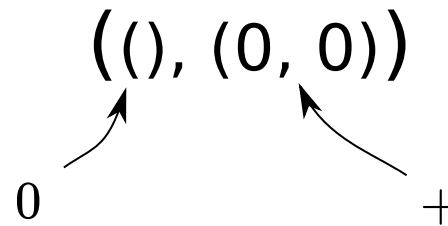
Definition

Binding signature = a family of lists of natural numbers.

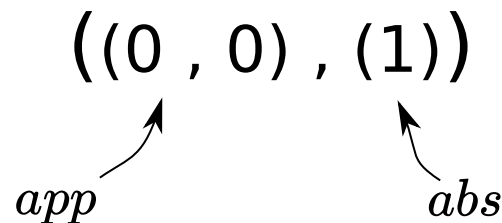
Each list specifies one operation in the syntax:

- 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, +:



Lambda calculus:



Initial semantics for binding signatures

The syntax $(0,+)$ is the initial algebra for the endofunctor:

$$F \mapsto F \times F + 1 + \text{Id}_{\text{Set}}$$

More generally, any binding signature gives rise to an endofunctor Σ .

Definition

Model = $(\Sigma + \text{Id}_{\text{Set}})$ -algebra

e.g. LCD is a model of the $(0,+)$ signature

Classical Theorem

The initial $(\Sigma + \text{Id}_{\text{Set}})$ -algebra of a binding signature Σ always exists.

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

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

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** Σ is a functorial assignment:

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

A **model of** Σ is a pair:

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

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** Σ is a functorial assignment:

$$\text{monad} \nearrow R \mapsto \Sigma(R)$$

A **model of** Σ is a pair:

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

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** Σ is a 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 := 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** Σ is a 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

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** Σ is a 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

monad := endofunctor with substitution

module over a monad := endofunctor with substitution

module morphism := natural transformation preserving substitution

Substitution and monads

Reminder:

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

$$LCD(f)(t) = t[x \mapsto f(x)]$$

Substitution and monads

Reminder:

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

$$LCD(f)(t) = t[\underbrace{x \mapsto f(x)}]$$

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

Substitution and monads

Reminder:

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

$$LCD(f)(t) = t[\underbrace{x \mapsto f(x)}]$$

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

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

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

Substitution and monads

Reminder:

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

$$LCD(f)(t) = t[\underbrace{x \mapsto f(x)}]$$

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

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

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

The triple $(LCD, \text{var}, \text{bind})$ is called a **monad**.

monad morphism = mapping preserving var and bind .

Monads

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

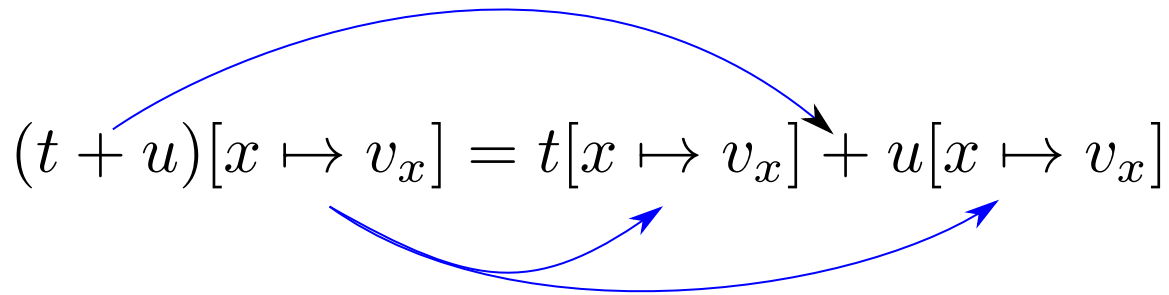
$$var(y)[x \mapsto u(x)] = u(y)$$

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

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

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

$LCD \times LCD$ supports
 LCD -substitution



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

+ commutes
with substitution



$+ : LCD \times LCD \rightarrow LCD$ is
a **module morphism**

Modules VS Monads

Monad

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

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

4. Substitution laws:

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

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

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

Modules VS Monads

~~Monad~~ **Module over a monad** B (e.g. $B \times B, 2, \dots$)

1. $M : \text{Set} \rightarrow \text{Set}$

$M(X) = \text{expressions taking variables in } X$

~~2. A collection of functions $(\text{var}_X : X \rightarrow M(X))_X$~~

3. For each function $u : X \rightarrow B(Y)$, a function $\text{bind}_u : M(X) \rightarrow M(Y)$

Parallel substitution

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

4. Substitution laws:

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

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

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

Building blocks for binding signatures

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

- R itself
- $M \times N$ for any modules M and N (in particular, $R \times R$)
- The **derivative of a module M** is the module M' defined by $M'(X) = M(X + \{\diamond\})$.

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

Syntactic operations are module morphisms

module morphism = maps commuting with substitution.

$$id_M : M \rightarrow M$$

$$0 : 1 \rightarrow LCD$$

$$+ : LCD \times LCD \rightarrow LCD$$

$$app : LCD \times LCD \rightarrow LCD$$

$$abs : LCD' \rightarrow LCD$$

The Big Picture again

A **1-signature** Σ is a functorial assignment:

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

monad \quad module over R

A **model of Σ** is a pair:

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

monad \quad module morphism

A **model morphism** $m : (R, \rho) \rightarrow (S, \sigma)$ is a 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.

Counter-example: the 1-signature $R \mapsto \mathcal{P} \circ R$



powerset endofunctor on Set

Examples of 1-signatures generating syntax

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

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

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: $R \mapsto R' + R \times R$

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, coproducts, and the trivial 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: lambda calculus modulo beta and eta.

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

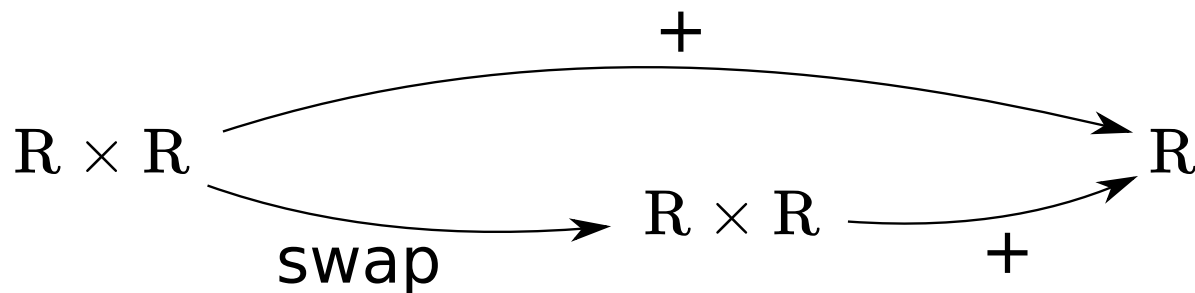
1-Signature: $R \mapsto R \times R$

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]: existence of an initial model for an inductively defined (with a specific syntax) set of possible equations.
- [AHLM CSL 2018]: "quotients" of algebraic 1-signatures generate a syntax (e.g. a binary commutative operation).

Review: Signatures with equations

- [Fiore-Hur 2010]: existence of an initial model for an inductively defined (with a specific syntax) set of possible equations.

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

- [AHLM CSL 2018]: "quotients" of algebraic 1-signatures generate a syntax (e.g. a binary commutative operation).

Review: Signatures with equations

- [Fiore-Hur 2010]: existence of an initial model for an inductively defined (with a specific syntax) set of possible equations.

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

- [AHLM CSL 2018]: "quotients" of algebraic 1-signatures generate a syntax (e.g. a binary commutative operation).

This work: more general equations (e.g. λ -calculus modulo $\beta\eta$).

Equations

Given a 1-signature Σ , a **Σ -equation** $A \rightrightarrows B$ is a functorial assignment

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

model of Σ parallel pair of module morphisms over R

A **2-signature** is a pair

$$(\Sigma, E)$$

1-signature set of Σ -equations

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

Algebraic 2-signatures

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

1. A "preserves pointwise epimorphisms"

(e.g., any "algebraic 1-signature")

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

Algebraic 2-signature:

(Σ, E)

algebraic 1-signature \nearrow \nwarrow set of **elementary**
 Σ -equations

Theorem

Syntax exists for any algebraic 2-signature

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

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

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

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

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 $\beta\eta$:

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

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: $t = \lambda x.(t x)$

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

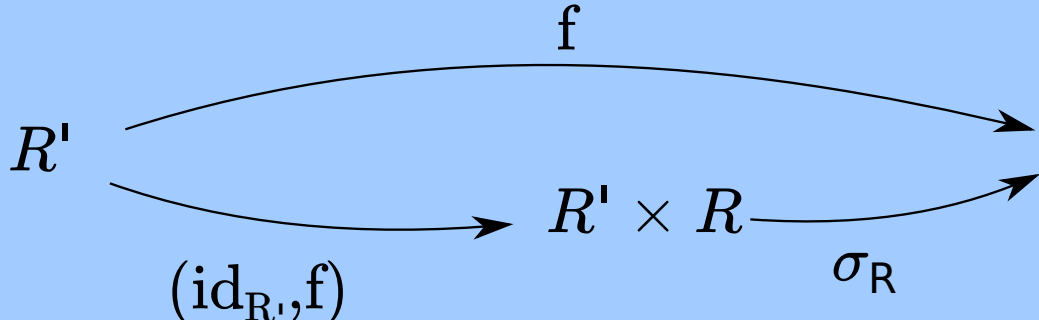
$$\begin{array}{ccccc}
 & \text{id}_{\mathbf{R}} & & & \\
 \mathbf{R} & \xrightarrow{\quad} & \mathbf{R} & & \\
 \text{R}t_1 \searrow & & \nearrow \text{abs} & & \\
 & \mathbf{R}' & & &
 \end{array}$$

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

Example: fixpoint operator

Definition [AHLM CSL 2018]

A **fixpoint operator** in a monad R is a module morphism $f : R' \rightarrow R$ s.t.

(1)  commutes.

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

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

Proposition [AHLM CSL 2018]

Fixpoint operators in $LC_{\beta\eta}$ are in one to one correspondance with fixpoint combinators (i.e. 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 $\beta\eta$

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

+

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

=

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

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

Example: free monoid

An algebraic 2-signature $(\Sigma_{\text{mon}}, E_{\text{mon}})$ for the free monoid monad $X \mapsto \coprod_n X^n$

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

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

$$\varepsilon : 1 \rightarrow \mathbf{R} \quad m : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$$

3 elementary Σ -equations:

A commutative diagram illustrating the associativity equation. It features three nodes: R^3 on the left, R on the right, and R^2 at the top center. There are two curved arrows from R^3 to R^2 : the top one is labeled $R \times m$ and the bottom one is labeled $m \times R$. From each R^2 node, a curved arrow labeled m points to the R node.

associativity

A commutative diagram illustrating the left unit equation. It features two nodes: R on the left and R on the right, with R^2 at the top center. A curved arrow labeled ϵ, id points from the left R to R^2 . A curved arrow labeled m points from R^2 to the right R . A curved arrow labeled id points directly from the left R to the right R .

left unit

A commutative diagram illustrating the right unit equation. It features two nodes: R on the left and R on the right, with R^2 at the top center. A curved arrow labeled id, ϵ points from the right R to R^2 . A curved arrow labeled m points from R^2 to the left R . A curved arrow labeled id points directly from the left R to the right R .

right unit

Our target: LCD

Syntax of the *differentiable* λ -calculus:

Simple terms $s, t \in \Lambda$

$s, t ::=$	x	}	λ -calculus
	$\lambda x. t$		
	$s \ t$		
	$Ds \cdot t$		
	$s + t$	}	free commutative monoid
	0		

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

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

Algebraic 1-signature for LCD

Syntax of the *differentiable* λ -calculus:

Simple terms $s, t \in \Lambda$

Corresponding 1-signature

$s, t ::=$	x	$\left. \begin{array}{l} \\ \\ \end{array} \right\}$	$\Sigma_{\text{LC}}(\mathbf{R}) = \mathbf{R}' + \mathbf{R} \times \mathbf{R}$
	$\lambda x. t$		
	$s \ t$		
	$\mathbf{D}s \cdot t$		$\mathbf{R} \mapsto \mathbf{R} \times \mathbf{R}$
	$s + t$	$\left. \begin{array}{l} \\ \end{array} \right\}$	$\Sigma_{\text{mon}}(\mathbf{R}) = 1 + \mathbf{R} \times \mathbf{R}$
	0		

Algebraic 1-signature for LCD

Syntax of the *differentiable* λ -calculus:

Simple terms $s, t \in \Lambda$

Corresponding 1-signature

$s, t ::=$	x	}	$\Sigma_{\text{LC}}(\mathbf{R}) = \mathbf{R}' + \mathbf{R} \times \mathbf{R}$
	$\lambda x. t$		
	$s \ t$		
	$\mathbf{D}s \cdot t$		$\mathbf{R} \mapsto \mathbf{R} \times \mathbf{R}$
	$s + t$	}	$\Sigma_{\text{mon}}(\mathbf{R}) = 1 + \mathbf{R} \times \mathbf{R}$
	0		

Resulting algebraic 1-signature:

$$\Sigma_{\text{LCD}}(\mathbf{R}) = \Sigma_{\text{LC}}(\mathbf{R}) + \mathbf{R} \times \mathbf{R} + \Sigma_{\text{mon}}(\mathbf{R})$$

Elementary equations for LCD

Commutative monoidal structure:

$$\begin{array}{ll} & s + t = t + s \\ E_{\text{mon}} \left\{ \begin{array}{l} s + (t + u) = (s + t) + u \\ 0 + t = t \\ t + 0 = t \end{array} \right. & \begin{array}{l} R \times R \Rightarrow R \\ R \times R \times R \Rightarrow R \\ R \Rightarrow R \\ R \Rightarrow R \end{array} \end{array}$$

Linearity:

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

...


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 (Σ, 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 (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)

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

\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)

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

\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)

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

\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)

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

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)

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

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)

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

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 2010]:

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)

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

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 2010]:

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 (Σ_{LC}, \emptyset)

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

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 2010]:

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 framework 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 framework to simply typed syntaxes.

Thank you!