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 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:

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

Differential λ-terms:

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

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:

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

variable

modulo α-renaming of x

Differential λ-terms:

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

neutral element for $+$
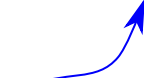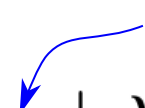
modulo commutativity

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:

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

variable

modulo α-renaming of x

Differential λ-terms:

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

neutral element for $+$

modulo commutativity

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

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:

variable

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

modulo α-renaming of x

Differential λ-terms:

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

neutral element for $+$

modulo commutativity

$$\Lambda^d = \mathsf{FreeCommutativeMonoid}(\Lambda^s)$$
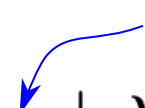
A syntax is specified by operations and equations.
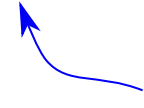
Syntax of LCD: version 1/3
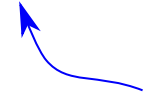
A **syntax** for the **_differentiable λ-calculus_** by **_mutual induction:_**

[Categorical Models for Simply Typed Resource Calculi]

Simple terms:

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

variable

modulo α-renaming of x

Differential λ-terms:

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

neutral element for $+$
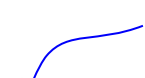
modulo commutativity

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

A syntax is specified by operations and equations.

But which ones are allowed ? What is the limit ?

Which operations/equations are allowed to specify a syntax ?

Can we avoid mutual induction ?

A stand-alone presentation of simple terms:

Simple terms:

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

Differential λ-terms:

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

Which operations/equations are allowed to specify a syntax ?

Can we avoid mutual induction ?

A stand-alone presentation of simple terms:

Simple terms:

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

as an operation: $\Lambda^s \times \mathsf{FreeCommutativeMonoid}(\Lambda^s) \to \Lambda^s$

Differential λ-terms:

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

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:

$$\Lambda^{\mathrm{d}} : \quad S,T \quad ::= \quad x \mid \lambda x.S \mid S\,T \mid \mathsf{D}S{\cdot}T$$

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

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 ?

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?

Recursion

Signature

Syntax
(initial model)

Category of Models
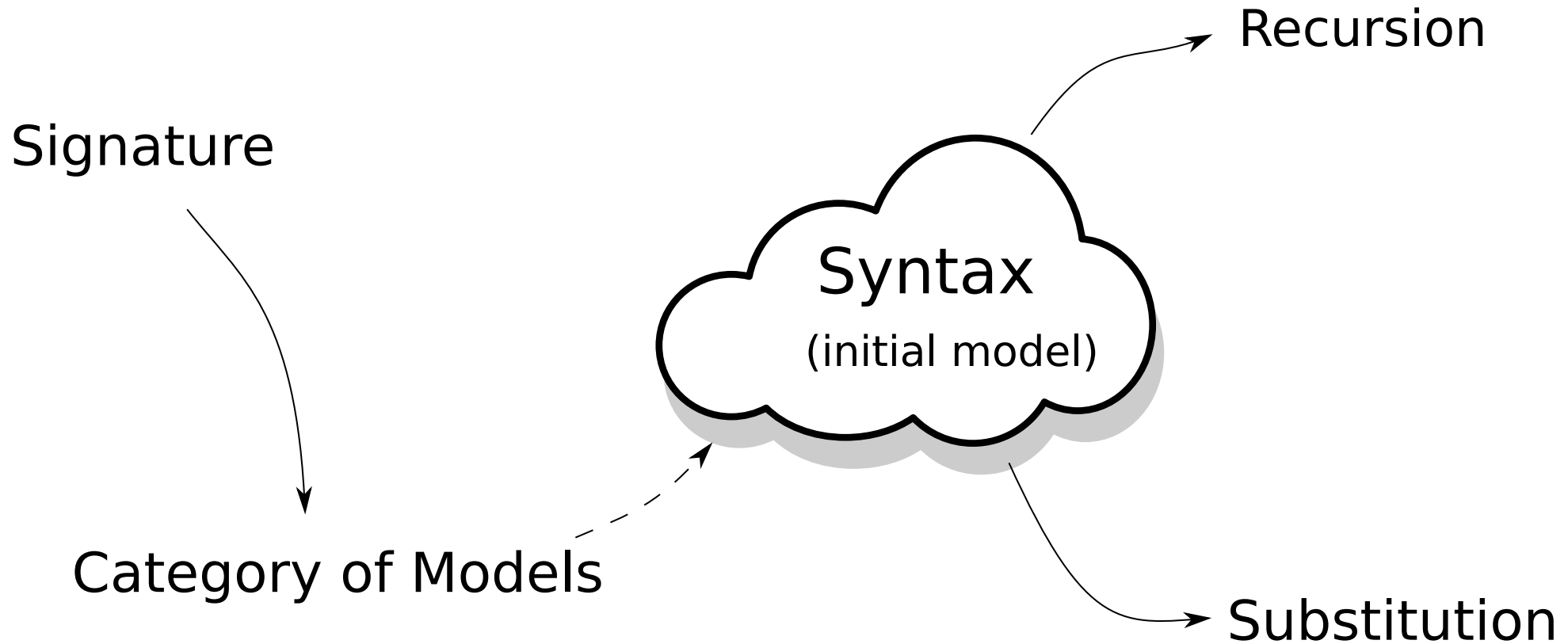
Substitution

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

Table of contents

Example: differential λ-calculus (last variant)

$$\Lambda^{\mathrm{d}} : \quad S,T \quad ::= \quad x \mid \lambda x.S \mid S\,T \mid \mathsf{D}S{\cdot}T$$
$$\mid\; 0 \;\mid\; S + T$$

Free variable indexing:

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

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

Categorical formulation of a term language

Example: differential λ-calculus (last variant)

$$\Lambda^{\mathrm{d}} : \quad S,T \quad ::= \quad x \mid \lambda x.\,S \mid S\,T \mid \mathsf{D}S\cdot T$$

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

Free variable indexing:

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

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

Free variable renaming:

$$\mathrm{LCD(f)} \;:\; \mathrm{LCD(X)} \rightarrow \quad \mathrm{LCD(Y)}$$
$$\mathrm{t} \quad \mapsto \quad \mathrm{t[x \mapsto f(x)]}$$

where $\quad \mathrm{f : X \rightarrow Y}$

Categorical formulation of a term language

Example: differential λ-calculus (last variant)

$$\Lambda^{\mathrm{d}} : \quad S,T \quad ::= \quad x \mid \lambda x.S \mid S\,T \mid \mathsf{D}S\cdot T$$
$$\mid\; 0 \;\mid\; S+T$$

Free variable indexing:

$$LCD : X \mapsto \{\text{terms taking free variables in } X\}$$
$$LCD(\emptyset) = \{0, \lambda z.z, \dots\}$$
$$LCD(\{x,y\}) = \{0, \lambda z.z, \dots, x, y, x+y, \dots\}$$

Free variable renaming:

$$
\begin{aligned}
\mathrm{LCD(f)} \;:\; \mathrm{LCD(X)} &\to \quad \mathrm{LCD(Y)} \\
\mathrm{t} &\mapsto \quad \mathrm{t[x \mapsto f(x)]}
\end{aligned}
\qquad \text{where} \quad \mathrm{f : X \to Y}
$$

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

Categorical formulation of a term language

Operations as natural transformations:

$$+ : \ LCD \times LCD \ \overset{\cdot}{\rightarrow} LCD$$

$$0 : \quad 1 \quad \overset{\cdot}{\rightarrow} LCD$$

$$\ldots$$

Variables as a natural transformation:

$$\mathrm{var} : \quad \mathrm{Id}_{\mathrm{Set}} \overset{\cdot}{\rightarrow} LCD$$

Putting all together:

$$LCD \times LCD + 1 + \mathrm{Id}_{\mathrm{Set}} \overset{\cdot}{\rightarrow} LCD$$

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, +:
$$\big((), (0, 0)\big)$$

$$0 \qquad\qquad +$$
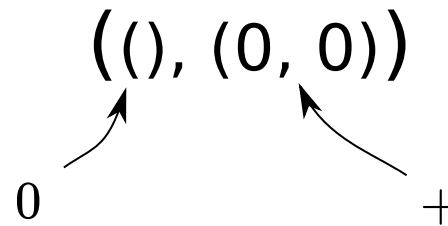
Lambda calculus:
$$\big((0, 0), (1)\big)$$

$$app \qquad\qquad abs$$
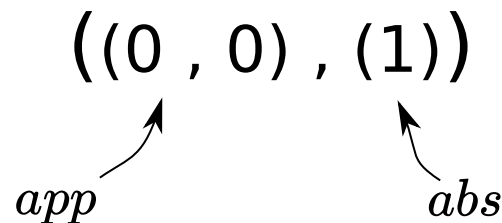
Initial semantics for binding signatures

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

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

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

Definition
Model = $(\Sigma + \mathrm{Id}_{\mathrm{Set}})$-algebra   e.g. LCD is a model of the (0,+) signature

Classical Theorem
The initial $(\Sigma + \mathrm{Id}_{\mathrm{Set}})$-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

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:

$$(R, \quad \rho : \Sigma(R) \to R)$$

$$
\begin{array}{rcl}
\text{monad} & := & \text{endofunctor with substitution} \\
\text{module over a monad} & := & \text{endofunctor with substitution} \\
\text{module morphism} & := & \text{natural transformation preserving substitution}
\end{array}
$$

Binding signatures $\hookrightarrow$ Our 1-signatures

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

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

monad

A **model of** Σ is a pair:

$$(R, \quad \rho : \Sigma(R) \to R)$$

$$
\begin{aligned}
\text{monad} \ &:= \ \text{endofunctor with substitution} \\
\text{module over a monad} \ &:= \ \text{endofunctor with substitution} \\
\text{module morphism} \ &:= \ \text{natural transformation preserving substitution}
\end{aligned}
$$

Binding signatures $\hookrightarrow$ Our 1-signatures

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

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

monad   module over R

A **model of** Σ is a pair:

$$(R, \quad \rho : \Sigma(R) \to R)$$

monad	:=	endofunctor with substitution
module over a monad	:=	endofunctor with substitution
module morphism	:=	natural transformation preserving substitution

Binding signatures $\hookrightarrow$ Our 1-signatures

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

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

monad   module over R

A **model of** Σ is a pair:

$$(R, \quad \rho : \Sigma(R) \to R)$$

monad

$$
\begin{aligned}
\text{monad} &:= \text{endofunctor with substitution} \\
\text{module over a monad} &:= \text{endofunctor with substitution} \\
\text{module morphism} &:= \text{natural transformation preserving substitution}
\end{aligned}
$$

Binding signatures $\hookrightarrow$ Our 1-signatures

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

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

monad   module over R

A **model of** Σ is a pair:

$$(R, \quad \rho : \Sigma(R) \to R)$$

monad   module morphism

monad	:=	endofunctor with substitution
module over a monad	:=	endofunctor with substitution
module morphism	:=	natural transformation preserving substitution

Substitution and monads

Reminder:

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

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

Substitution and monads

Reminder:

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

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

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

Substitution and monads

Reminder:

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

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

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

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

$$
\begin{array}{rcll}
\mathrm{bind}_{\mathrm{f}} & : & LCD(\mathrm{X}) & \to & LCD(\mathrm{Y}) \\
& & \mathrm{t} & \mapsto & \mathrm{t}[\mathrm{x} \mapsto \mathrm{f}(\mathrm{x})]
\end{array}
$$

where $\quad \mathrm{f} : \mathrm{X} \to LCD(\mathrm{Y})$

Substitution and monads

Reminder:

- $LCD(X) = \{$ differential λ-terms taking free variables in X $\}$

- Variables induce a natural transformation $\mathrm{var} : \mathrm{Id}_{\mathrm{Set}} \to LCD$

- **Variable renaming** by functoriality:

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

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

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

$$
\begin{aligned}
\mathrm{bind}_{\mathrm{f}} \;:\; LCD(\mathrm{X}) &\;\to\; LCD(\mathrm{Y}) \\
\mathrm{t} &\;\mapsto\; \mathrm{t}[\mathrm{x} \mapsto \mathrm{f}(\mathrm{x})]
\end{aligned}
\qquad \text{where} \quad \mathrm{f} : \mathrm{X} \to LCD(\mathrm{Y})
$$

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

monad morphism $=$ mapping preserving var and bind.

Monads

1. $LCD : \mathrm{Set} \to \mathrm{Set}$

2. A collection of functions $(\mathrm{var_X} : X \to LCD(X))_X$

   Variables are expressions

3. For each function $\mathrm{u} : \mathrm{X} \to \mathrm{LCD(Y)}$, a function $\mathrm{bind_u} : \mathrm{LCD(X)} \to \mathrm{LCD(Y)}$

   Parallel substitution

   Notation: $\mathbf{bind_u(t) = t[x \mapsto u(x)]}$

4. Monadic laws:

$$\mathrm{var(y)[x \mapsto u(x)] = u(y)}$$
$$\mathrm{t[x \mapsto var(x)] = t}$$
$$\mathrm{t[x \mapsto f(x)][y \mapsto g(y)] = t[x \mapsto f(x)[y \mapsto g(y)]\,]}$$

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

+ commutes with substitution $\quad\rightsquigarrow\quad$ $+ : LCD \times LCD \to LCD$ is a **module morphism**

Modules VS Monads

Monad

1. $B : \mathrm{Set} \to \mathrm{Set}$

2. A collection of functions $(\mathrm{var}_X : X \to B(X))_X$

   Variables are expressions

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

   Parallel substitution

   Notation: $\mathbf{bind_u(t) = t[x \mapsto u(x)]^B}$
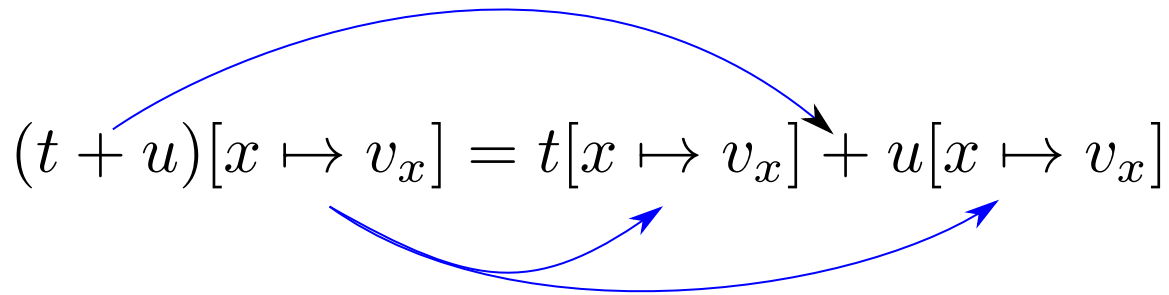
4. Substitution laws:

$$\mathrm{var}(y)[x \mapsto u(x)]^B = u(y)$$
$$t[x \mapsto \mathrm{var}(x)]^B = t$$
$$t[x \mapsto f(x)]^B[y \mapsto g(y)]^B = t[x \mapsto f(x)[y \mapsto g(y)]^B]^B$$

Modules VS Monads

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

1. $M : \mathrm{Set} \to \mathrm{Set}$

   $M(X)$ = expressions taking variables in X

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

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

   Parallel substitution

   Notation: $\mathbf{bind_u(t)} = \mathbf{t[x \mapsto u(x)]^M}$

4. Substitution laws:

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

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

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

Building blocks for binding signatures

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

- R itself

- M x N for any modules M and N (in particular, R x 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.

$$id_M : M \to M$$

$$0 : 1 \to LCD$$

$$+ : LCD \times LCD \to LCD$$

$$app : LCD \times LCD \to LCD$$

$$abs : LCD' \to LCD$$

The Big Picture again

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

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

monad   module over R

A **model of** Σ is a pair:

$$(R, \quad \rho : \Sigma(R) \to R)$$

monad   module morphism

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

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

Syntax

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 $R \mapsto \mathscr{P} \circ R$

powerset endofunctor on Set

Examples of 1-signatures generating syntax

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

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

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

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

- **lambda calculus**:

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

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

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

Can we generalize this pattern?

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:

$$\text{Binding signatures} \hookrightarrow \text{Our 1-signatures}$$

Question: Can we enforce some equations in the syntax ?

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

Table of contents

Example: a commutative binary operation

Specification of a binary operation

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

Model: $(R , \; + : R \times R \to R)$

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: $(R,\ + : R \times R \to R)$ 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

Specification of a commutative binary operation

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

Model: $(R ,\ + : R \times R \to R)$ 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:
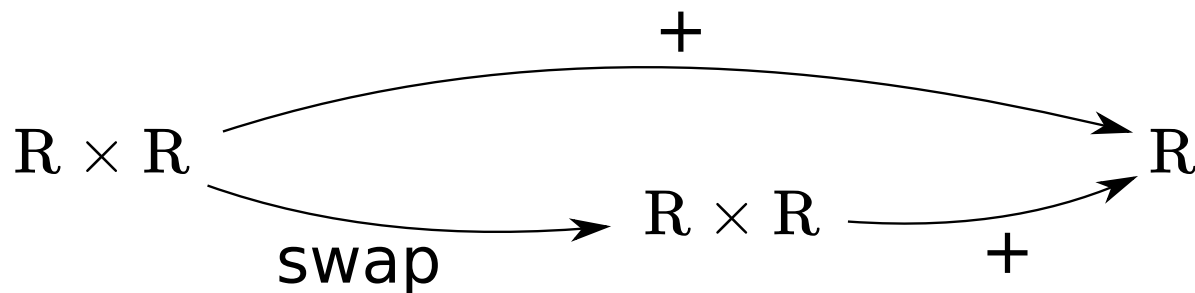
$$
\begin{array}{ccc}
 & \xrightarrow{\quad + \quad} & \\
R \times R & & R \\
 & \xrightarrow[\text{swap}]{} R \times R \xrightarrow[+]{} &
\end{array}
$$

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

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

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

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

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

$$\text{app} : \text{LCD} \times \text{FreeCommutativeMonoid(LCD)} \to \text{LCD}$$

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

Equations

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

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

model of Σ

parallel pair of module

morphisms over R

A **2-signature** is a pair

$$(\Sigma, \mathbf{E})$$

1-signature   set of Σ-equations

model of a 2-signature $(\Sigma, \mathbf{E})$:

- a model R of Σ

- s.t. $\forall$ (A $\Rrightarrow$ B) $\in$ E, the two morphisms $A(R) \Rrightarrow B(R)$ are equal

Algebraic 2-signatures

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

1. A "preserves pointwise epimorphisms"

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

2. B is of the form R ↦ R'...' (e.g. R ↦ R)

Algebraic 2-signature:

$$(\Sigma, E)$$

algebraic 1-signature

set of **elementary**
Σ-equations

> **Theorem**
> Syntax exists for any algebraic 2-signature

Example: λ-calculus modulo βη

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

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

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

$$\mathrm{app} : R \times R \to R \qquad \mathrm{abs} : R' \to R$$

β-equation: $(\lambda x.t)\, u = \underbrace{t[x \mapsto u]}_{\sigma_R(t,u)}$ **η-equation**: $t = \lambda x.(t\, x)$

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

Example: λ-calculus modulo βη

The algebraic 2-signature $(\Sigma_{\mathrm{LC}\beta\eta}, \mathrm{E}_{\mathrm{LC}\beta\eta})$ of λ-calculus modulo βη:

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

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

$$\mathrm{app} : \mathrm{R} \times \mathrm{R} \to \mathrm{R} \qquad \mathrm{abs} : \mathrm{R'} \to \mathrm{R}$$

β-equation: $(\lambda\mathrm{x}.\mathrm{t})\,\mathrm{u} = \underbrace{\mathrm{t}[\mathrm{x} \mapsto \mathrm{u}]}_{\color{blue}\sigma_{\mathrm{R}}(\mathrm{t},\mathrm{u})}$ 	**η-equation**: $\mathrm{t} = \lambda\mathrm{x}.(\mathrm{t}\,\mathrm{x})$
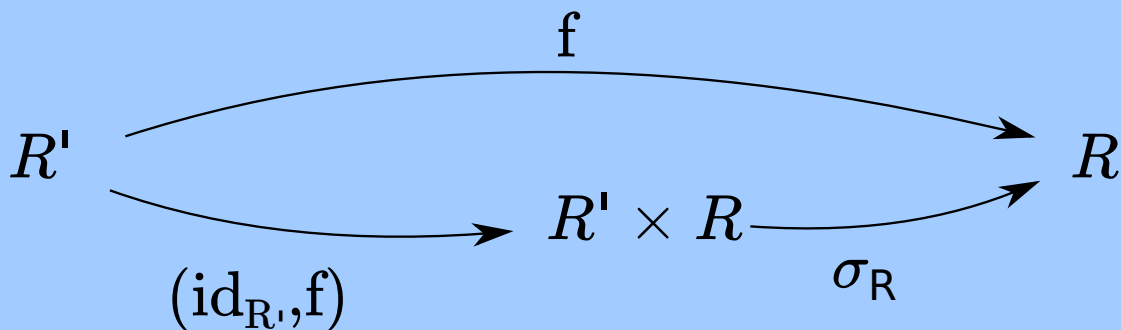
$$\mathbf{E}_{\mathbf{LC\beta\eta}} = \{\ \beta\text{-equation},\ \eta\text{-equation}\ \}$$

Example: fixpoint operator

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

(1)

$$R' \xrightarrow{\quad f \quad} R \qquad \text{commutes.}$$

with lower path $R' \xrightarrow{(\mathrm{id}_{R'}, f)} R' \times R \xrightarrow{\sigma_R} R$

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

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

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

Combining algebraic 2-signatures

Algebraic 2-signatures can be combined:

fixpoint operator

λ-calculus modulo βη

$$(\Sigma_{\mathrm{fix}}, E_{\mathrm{fix}}) \qquad + \qquad (\Sigma_{\mathrm{LCβη}}, E_{\mathrm{LCβη}})$$

$$=$$

$$(\Sigma_{\mathrm{fix}} + \Sigma_{\mathrm{LCβη}} \quad , \quad E_{\mathrm{fix}} \cup E_{\mathrm{LCβη}})$$

λ-calculus modulo βη with an explicit fixpoint operator

Example: free commutative monoid

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

monad:
$$\Sigma_{\mathrm{mon}}(R) := 1 + R \times R$$

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

$$0 : 1 \to R \qquad + : R \times R \to R$$

Example: free commutative monoid

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

monad: $$\boldsymbol{\Sigma}_{\mathrm{mon}}(R) := 1 + R \times R$$

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

$$0 : 1 \to R \qquad + : R \times R \to R$$

4 elementary Σ-equations:

Syntax of the *differentiable λ-calculus*:

Simple terms $s, t \in \Lambda$

$$
\begin{array}{rcl}
s,t & ::= & x \\
& | & \lambda x.t \\
& | & s\,t \\
& | & Ds \cdot t \\
& | & s + t \\
& | & 0
\end{array}
$$

λ-calculus

free commutative monoid

and (bi)linearity of constructors with respect to +:
$$\lambda x.(s+t) = \lambda x.s + \lambda x.t \qquad \ldots$$

Algebraic 1-signature for LCD

Syntax of the *differentiable λ-calculus*:

Simple terms $s,t \in \Lambda$ Corresponding 1-signature

$$
\begin{aligned}
s,t \ ::= \ & x \\
\mid \ & \lambda x.t \\
\mid \ & s\ t \\
\mid \ & Ds \cdot t \\
\mid \ & s + t \\
\mid \ & 0
\end{aligned}
$$

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

$$R \mapsto R \times R$$

$$\Sigma_{\mathrm{mon}}(R) = 1 + R \times R$$

Syntax of the *differentiable λ-calculus*:

Simple terms $s, t \in \Lambda$ Corresponding 1-signature

$$
\begin{aligned}
s, t \; ::= \; & x \\
| \; & \lambda x.t \\
| \; & s\,t \\
| \; & Ds \cdot t \\
| \; & s + t \\
| \; & 0
\end{aligned}
$$

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

$R \mapsto R \times R$

$\Sigma_{\mathrm{mon}}(R) = 1 + R \times R$

Resulting algebraic 1-signature: $\Sigma_{\mathrm{LCD}}(R) = \Sigma_{\mathrm{LC}}(R) + R \times R + \Sigma_{\mathrm{mon}}(R)$

Elementary equations for LCD

Commutative monoidal structure:

$$E_{mon} \begin{cases} s + t = t + s & R \times R \rightrightarrows R \\ s + (t + u) = (s + t) + u & R \times R \times R \rightrightarrows R \\ 0 + t = t & R \rightrightarrows R \\ t + 0 = t & R \rightrightarrows R \end{cases}$$

Linearity:

$$\lambda x.(s+t) = \lambda x.s + \lambda x.t \qquad R \times R \rightrightarrows R$$

$$D(s+t) \cdot u = Ds \cdot u + Dt \cdot u \qquad R \times R \times R \rightrightarrows R$$

$$Ds \cdot (t+u) = Ds \cdot t + Ds \cdot u \qquad R \times R \times R \rightrightarrows R$$

$$\cdots$$

Table of contents

Principle of recursion

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

Recipe for constructing "by recursion" a monad morphism:

$$f : R \to S$$

initial model of a 2-signature $(\Sigma, \mathbb{E})$

Principle of recursion

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

Recipe for constructing "by recursion" a monad morphism:

$$f : R \to S$$

initial model of a 2-signature $(\Sigma, \mathbb{E})$

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

Principle of recursion

Recursion on the syntax ≃ Initiality in the category of models

Recipe for constructing "by recursion" a monad morphism:

$$f : R \to S$$

initial model of a 2-signature (Σ, E)

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

    $\Rightarrow$ 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:

$$f : \mathrm{R} \to \mathrm{S}$$

initial model of a 2-signature (Σ, E)

1. Give a module morphism $\mathrm{s} : \Sigma(\mathrm{S}) \to \mathrm{S}$

   $\Rightarrow$ 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:

$$f : R \to S$$

initial model of a 2-signature (Σ, E)

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

    $\Rightarrow$ 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 ≃ Initiality in the category of models

Recipe for constructing "by recursion" a monad morphism:

$$\mathrm{f} : \mathrm{R} \to \mathrm{S}$$

initial model of a 2-signature (Σ, E)

1. Give a module morphism $\mathrm{s} : \Sigma(\mathrm{S}) \to \mathrm{S}$

    $\Rightarrow$ 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 $\mathrm{R} \to \mathrm{S}$ $\Rightarrow$ monad morphism $\mathrm{R} \to \mathrm{S}$

LC = initial model of $(\Sigma_{\mathrm{LC}}, \varnothing)$ $\qquad\qquad \Sigma_{\mathrm{LC}}(\mathrm{R}) = \mathrm{R} \times \mathrm{R} + \mathrm{R'}$

$\mathcal{P}$ = power set monad

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

$$\mathrm{fv}(\mathrm{app}(\mathrm{t,u})) = \mathrm{fv}(\mathrm{t}) \cup \mathrm{fv}(\mathrm{u}) \qquad\qquad \mathrm{fv}(\mathrm{abs}(\mathrm{t})) = \mathrm{fv}(\mathrm{t}) \setminus \{\diamond\}$$

LC = initial model of $(\Sigma_{\mathrm{LC}}, \varnothing)$ $\qquad\qquad$ $\Sigma_{\mathrm{LC}}(\mathrm{R}) = \mathrm{R} \times \mathrm{R} + \mathrm{R}'$

$\mathcal{P}$ = power set monad

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

$$\mathrm{fv}(\mathrm{app}(\mathrm{t},\mathrm{u})) = \mathrm{fv}(\mathrm{t}) \cup \mathrm{fv}(\mathrm{u}) \qquad\qquad \mathrm{fv}(\mathrm{abs}(\mathrm{t})) = \mathrm{fv}(\mathrm{t}) \setminus \{\diamond\}$$

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

$$\cup : \mathcal{P} \times \mathcal{P} \to \mathcal{P} \qquad\qquad\qquad \_\setminus\{\diamond\} : \mathcal{P}' \to \mathcal{P}$$

LC = initial model of $(\Sigma_{\mathrm{LC}}, \varnothing)$ $\qquad\qquad\qquad$ $\Sigma_{\mathrm{LC}}(\mathrm{R}) = \mathrm{R} \times \mathrm{R} + \mathrm{R}'$

$\mathcal{P}$ $\;$ = power set monad

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

$$\mathrm{fv}(\mathrm{app(t,u)}) = \mathrm{fv(t)} \cup \mathrm{fv(u)} \qquad\qquad \mathrm{fv}(\mathrm{abs(t)}) = \mathrm{fv(t)} \setminus \{\diamond\}$$

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

$$\cup : \mathcal{P} \times \mathcal{P} \to \mathcal{P} \qquad\qquad\qquad\qquad \_\setminus\{\diamond\} : \mathcal{P}' \to \mathcal{P}$$

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

Example: Translating λ-calculus with fixpoint

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

    λ-calculus modulo βη with a fixpoint operator $\text{fix} : \text{LC}_{\beta\eta\text{fix}}' \to \text{LC}_{\beta\eta\text{fix}}$

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

    λ-calculus modulo βη

monad morphism

Definition of a translation $f : \text{LC}_{\beta\eta\text{fix}} \to \text{LC}_{\beta\eta}$ **s.t.**

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

a chosen fixpoint combinator

44 / 50

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

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

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

    λ-calculus modulo βη

monad morphism

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

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

a chosen fixpoint combinator

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

$$app, abs \qquad\qquad \hat{Y} : LC_{\beta\eta}' \rightarrow LC_{\beta\eta}$$

$$t \mapsto app(Y,abs(t))$$

Example: Translating λ-calculus with fixpoint

LC$_{βηfix}$ = initial model of $(\Sigma_{\mathrm{LC}\beta\eta}, \mathrm{E}_{\mathrm{LC}\beta\eta}) + (\Sigma_{\mathrm{fix}}, \mathrm{E}_{\mathrm{fix}})$

    λ-calculus modulo βη with a fixpoint operator $\mathrm{fix} : \mathrm{LC}_{\beta\eta fix}' \to \mathrm{LC}_{\beta\eta fix}$

LC$_{βη}$ = initial model of $(\Sigma_{\mathrm{LC}\beta\eta}, \mathrm{E}_{\mathrm{LC}\beta\eta})$

    λ-calculus modulo βη

monad morphism

Definition of a translation $\mathrm{f} : \mathrm{LC}_{βηfix} \to \mathrm{LC}_{βη}$ **s.t.**

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

a chosen fixpoint combinator

$\Rightarrow$ make LC$_{βη}$ a model of $(\Sigma_{\mathrm{LC}\beta\eta}, \mathrm{E}_{\mathrm{LC}\beta\eta}) + (\Sigma_{\mathrm{fix}}, \mathrm{E}_{\mathrm{fix}})$:

$\mathrm{app, abs}$ $\hat{\mathrm{Y}} : \mathrm{LC}_{βη}' \to \mathrm{LC}_{βη}$

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

Initiality of $\mathrm{LC}_{βηfix} \Rightarrow \mathrm{f} : \mathrm{LC}_{βηfix} \to \mathrm{LC}_{βη}$

LC = initial model of $(\Sigma_{\mathrm{LC}}, \varnothing)$ $\qquad\qquad \Sigma_{\mathrm{LC}}(\mathrm{R}) = \mathrm{R} \times \mathrm{R} + \mathrm{R}'$

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

$$\mathrm{s}(\mathrm{app}(\mathrm{t,u})) = 1 + \mathrm{s}(\mathrm{t}) + \mathrm{s}(\mathrm{u}) \qquad\qquad \mathrm{s}(\mathrm{abs}(\mathrm{t})) = 1 + \mathrm{s}(\mathrm{t})$$

LC = initial model of $(\Sigma_{\mathrm{LC}}, \varnothing)$ $\qquad\qquad\qquad \Sigma_{\mathrm{LC}}(R) = R \times R + R'$

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

$$s(\mathrm{app}(t,u)) = 1 + s(t) + s(u) \qquad\qquad s(\mathrm{abs}(t)) = 1 + s(t)$$

$\mathbb{N}$ **is not a monad !**

Example: Computing the size of a term

LC = initial model of $(\Sigma_{\mathrm{LC}}, \varnothing)$ $\qquad\qquad\qquad \Sigma_{\mathrm{LC}}(R) = R \times R + R'$

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

$$s(\mathrm{app(t,u)}) = 1 + s(t) + s(u) \qquad\qquad s(\mathrm{abs(t)}) = 1 + s(t)$$

$\mathbb{N}$ **is not a monad !**

Solution [CSL AHLM 2010]:
continuation monad $C(X) = \mathbb{N}^{(\mathbb{N}^X)}$
1. define $\quad f : \mathrm{LC} \to C$ by recursion
2. deduce $\;\; s : \mathrm{LC} \to \mathbb{N}$

LC = initial model of $(\Sigma_{\mathrm{LC}}, \varnothing)$ $\Sigma_{\mathrm{LC}}(\mathrm{R}) = \mathrm{R} \times \mathrm{R} + \mathrm{R}'$

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

$$s(app(t,u)) = 1 + s(t) + s(u) \qquad s(abs(t)) = 1 + s(t)$$

   $\mathbb{N}$ **is not a monad !**

Solution [CSL AHLM 2010]:   continuation monad $C(X) = \mathbb{N}^{(\mathbb{N}^X)}$
1. define $f : \mathrm{LC} \to \mathrm{C}$ by recursion
2. deduce $s : \mathrm{LC} \to \mathbb{N}$

                                  affects an arbitrary size to each variable

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

Example: Computing the size of a term

LC = initial model of $(\Sigma_{\mathrm{LC}}, \varnothing)$ $\qquad\qquad$ $\Sigma_{\mathrm{LC}}(\mathrm{R}) = \mathrm{R} \times \mathrm{R} + \mathrm{R'}$

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

$$s(\mathrm{app}(t,u)) = 1 + s(t) + s(u) \qquad\qquad s(\mathrm{abs}(t)) = 1 + s(t)$$

$\mathbb{N}$ **is not a monad !**

Solution [CSL AHLM 2010]:   continuation monad $C(X) = \mathbb{N}^{(\mathbb{N}^X)}$

1. define $\quad f : \mathrm{LC} \to C$ by recursion
2. deduce $\ s : \mathrm{LC} \to \mathbb{N}$

affects an arbitrary size to each variable

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

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

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!