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

Arity/Signature

Operation/Construction

This talk: give a *discipline* for specifying syntaxes

Motivating example: dLC

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

- explicitly involves **equations** e.g. $\mathrm{s}+\mathrm{t} = \mathrm{t}+\mathrm{s}$
- specifically taylored: (not an *instance* of a general framework/scheme)

  inductive definition of a set + ad-hoc structure
  e.g. **unary substitution**

Our proposal $=$ a discipline for presenting syntaxes

- signature = operations + equations
- [Fiore-Hure 2010]: alternative approach, for simply typed syntaxes

$\Rightarrow$ our approach explicitly relies on monads and modules (untyped case).

Motivating example: dLC

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

- explicitly involves **equations** e.g. $\mathrm{s}+\mathrm{t} = \mathrm{t}+\mathrm{s}$
- specifically taylored: (not an *instance* of a general framework/scheme)

  inductive definition of a set + ad-hoc structure
  e.g. **unary substitution**

Our proposal $=$ a discipline for presenting syntaxes

- signature = operations + equations
- [Fiore-Hure 2010]: alternative approach, for simply typed syntaxes

$\Rightarrow$ our approach explicitly relies on monads and modules (untyped case).

Syntax of dLC: [Ehrhard-Regnier 2003]

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

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

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

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

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

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

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

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

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

$$(s)t$$

belongs to Δ_{k+1}.

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

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

Syntax of dLC: [Ehrhard-Regnier 2003]

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

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

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

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

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

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

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

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

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

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

belongs to Δ_{k+1}.

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

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

Syntax of dLC: [BEM 2010]

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

[Bucciarelli-Ehrhard-Manzonetto 2010]

Simple terms:

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

Differential λ-terms:

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

Syntax of dLC: [BEM 2010]

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

[Bucciarelli-Ehrhard-Manzonetto 2010]

Simple terms:

variable

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

modulo α-renaming of x

Differential λ-terms:

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

neutral element for $+$

modulo commutativity

Syntax of dLC: [BEM 2010]

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

[Bucciarelli-Ehrhard-Manzonetto 2010]

Simple terms:

variable

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

modulo α-renaming of x

Differential λ-terms:

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

neutral element for $+$

modulo commutativity

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

Syntax of dLC: [BEM 2010]

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

[Bucciarelli-Ehrhard-Manzonetto 2010]

Simple terms:

variable

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

modulo α-renaming of x

Differential λ-terms:

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

neutral element for $+$

modulo commutativity

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

Syntax: specified by operations and equations.

Syntax of dLC: [BEM 2010]

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

[Bucciarelli-Ehrhard-Manzonetto 2010]

Simple terms:

variable

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

modulo α-renaming of x

Differential λ-terms:

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

neutral element for $+$

modulo commutativity

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

Syntax: specified by operations and equations.

But which ones are allowed ? What is the limit ?

Syntax of dLC: Our version

Which operations/equations are allowed to specify a syntax ?

A stand-alone presentation of differential λ-terms:

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

Differential λ-terms:

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

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

neutral element for $+$

modulo commutativity and associativity

Macros in [BEM 2010]:

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

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

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

Syntax of dLC: Conclusion

How can we compare these different versions ?

In which sense are they syntaxes ?

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

Syntax of dLC: Conclusion

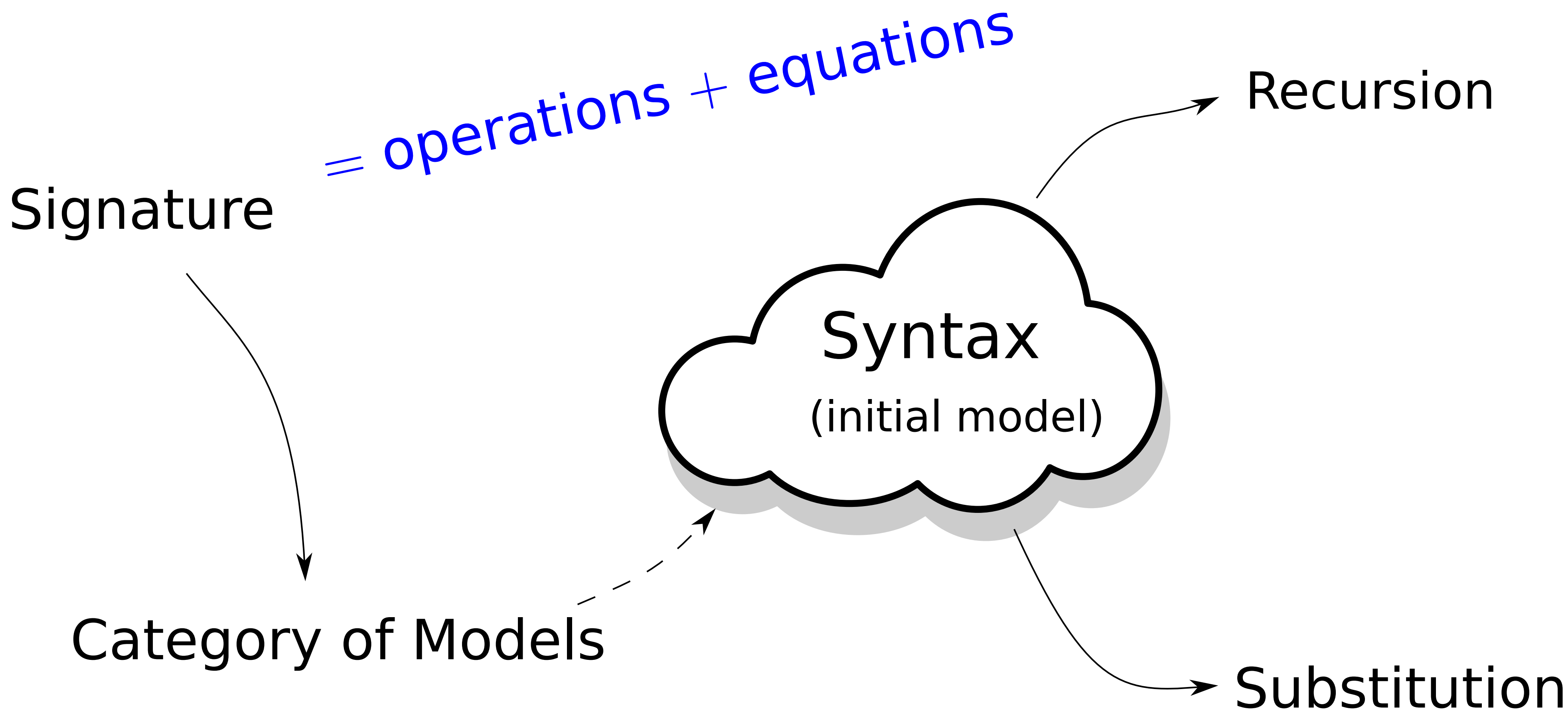
How can we compare these different versions ?

In which sense are they syntaxes ?

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

What is a syntax ?

What is a syntax?



generates a syntax = existence of the initial model

Table of contents

Table of contents

Substitution and monads

Example: differential λ-calculus

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

Free variable indexing:

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

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

Substitution and monads

Example: differential λ-calculus

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

Free variable indexing:

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

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

Parallel substitution:

Substitution and monads

Example: differential λ-calculus

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

Free variable indexing:

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

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

Parallel substitution:

$$\begin{array}{rcll} \mathrm{bind_f} & : dLC(\mathrm{X}) & \rightarrow & dLC(\mathrm{Y}) \\ & \mathrm{t} & \mapsto & \color{blue}{\mathrm{t}[\mathrm{x} \mapsto \mathrm{f}(\mathrm{x})]} \end{array}$$

where $\mathrm{f} : \mathrm{X} \rightarrow \color{red}{dLC(\mathrm{Y})}$

$\Rightarrow$ (dLC, var_X : X ⊂ dLC(X) , bind) $=$ **monad on $\mathbf{Set}$**

Substitution and monads

Example: differential λ-calculus

$$\begin{array}{rrcl} \Lambda^{\mathrm{d}} : & S, T & ::= & x \mid \lambda x.\, S \mid S\, T \mid \mathsf{D} S \cdot T \\ & & & \mid\ 0 \ \mid\ S + T \end{array}$$

Free variable indexing:

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

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

Parallel substitution:

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

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

$\Rightarrow$ (dLC, var_X : X ⊂ dLC(X) , bind) $=$ **monad on $\mathbf{Set}$**

monad morphism $=$ mapping preserving variables and substitutions.

Preview: Operations are module morphisms

+ commutes with substitution

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

Categorical formulation

$dLC \times dLC$ supports dLC-substitution $\rightsquigarrow$ $dLC \times dLC$ is a **module over $\boldsymbol{dLC}$**

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

Building blocks for specifying operations

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

- R itself

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

  $\mathrm{e.g.\ R \times R}$:

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

- M' = **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

operations = **module morphisms** = maps commuting with substitution.

$$\mathrm{app} : \mathrm{dLC} \times \mathrm{dLC} \to \mathrm{dLC} \qquad 0 : 1 \to \mathrm{dLC}$$

$$\mathrm{abs} : \mathrm{dLC'} \to \mathrm{dLC} \qquad + : \mathrm{dLC} \times \mathrm{dLC} \to \mathrm{dLC}$$

$$\begin{aligned} \mathrm{abs}_X : \mathrm{dLC}(\mathrm{X} \textstyle\coprod \{\diamond\}) &\to \mathrm{dLC}(X) \\ t &\mapsto \lambda\diamond.t \end{aligned}$$

Syntactic operations are module morphisms

operations = module morphisms = maps commuting with substitution.

$$\mathrm{app} : \mathrm{dLC} \times \mathrm{dLC} \to \mathrm{dLC} \qquad 0 : 1 \to \mathrm{dLC}$$

$$\mathrm{abs} : \mathrm{dLC'} \to \mathrm{dLC} \qquad + : \mathrm{dLC} \times \mathrm{dLC} \to \mathrm{dLC}$$

$$\begin{aligned} \mathrm{abs}_X : \mathrm{dLC}(X \textstyle\coprod \{\diamond\}) &\to \mathrm{dLC}(X) \\ t &\mapsto \lambda\diamond.t \end{aligned}$$

Combining operations into a single one using disjoint union

$$[\mathrm{app}, \mathrm{abs}] : (\mathrm{dLC} \times \mathrm{dLC}) \textstyle\coprod \mathrm{dLC'} \to \mathrm{dLC}$$

$$[0, +] : 1 \textstyle\coprod (\mathrm{dLC} \times \mathrm{dLC}) \to \mathrm{dLC}$$

1-signatures and their models

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

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

Example: (0,+)

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

A **model of Σ** is a pair:

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

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

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

A **model morphism** m : (R,ρ) → (S,σ) $=$ monad morphism commuting with the module morphism:

$$\begin{tikzcd}
\Sigma(R) \arrow[r, "\rho"] \arrow[d, "\Sigma(m)"'] & R \arrow[d, "m"] \\
\Sigma(S) \arrow[r, "\sigma"'] & S
\end{tikzcd}$$

1-signatures and their models

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

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

monad

A **model of Σ** is a pair:

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

Example: (0,+)

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

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

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

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

$$\begin{tikzcd} \Sigma(R) \arrow[r, "\rho"] \arrow[d, "\Sigma(m)"'] & R \arrow[d, "m"] \\ \Sigma(S) \arrow[r, "\sigma"'] & S \end{tikzcd}$$

1-signatures and their models

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

$$\underset{\text{monad}}{R} \mapsto \underset{\text{module over } R}{\Sigma(R)}$$

A **model of Σ** is a pair:

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

Example: (0,+)

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

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

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

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

$$\begin{tikzcd}
\Sigma(R) \arrow[r, "\rho"] \arrow[d, "\Sigma(m)"'] & R \arrow[d, "m"] \\
\Sigma(S) \arrow[r, "\sigma"'] & S
\end{tikzcd}$$

1-signatures and their models

A **1-signature** Σ = 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

Example: (0,+)

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

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

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

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

$$\begin{tikzcd}
\Sigma(R) \arrow[r, "\rho"] \arrow[d, "\Sigma(m)"'] & R \arrow[d, "m"] \\
\Sigma(S) \arrow[r, "\sigma"'] & S
\end{tikzcd}$$

1-signatures and their models

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

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

monad

module over R

Example: (0,+)

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

A **model of Σ** is a pair:

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

monad

module morphism

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

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

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

$$\begin{tikzcd}
\Sigma(R) \arrow[r, "\rho"] \arrow[d, "\Sigma(m)"'] & R \arrow[d, "m"] \\
\Sigma(S) \arrow[r, "\sigma"'] & S
\end{tikzcd}$$

Syntax

Definition

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

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

Answer: No.

Syntax

Definition

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

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

Answer: No.

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

powerset endofunctor on Set

Examples of 1-signatures generating syntax

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

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

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

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

- **lambda calculus**:

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

  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?

Initial semantics for algebraic 1-signatures

Theorem [Hirschowitz & Maggesi 2007]

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

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

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

Question: Can we enforce some equations in the syntax ?

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

Quotients of algebraic 1-signatures

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

Theorem [AHLM CSL 2018]

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

Examples:

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

$$\mathrm{app} : \mathrm{dLC} \times \mathrm{FreeCommutativeMonoid}(\mathrm{dLC}) \to \mathrm{dLC}$$

Quotients of algebraic 1-signatures

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

Theorem [AHLM CSL 2018]

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

Examples:

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

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

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

- associativity of $+$
- linearity of the operations

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

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

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

$$\begin{tikzcd} \mathrm{R} \times \mathrm{R} \arrow[rr, bend left, "+"] \arrow[r, "\text{swap}"'] & \mathrm{R} \times \mathrm{R} \arrow[r, "+"'] & \mathrm{R} \end{tikzcd}$$

Equations

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

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

e.g. commutativity:

$$R \mapsto \left(R \times R \underset{+\circ swap}{\overset{+}{\rightrightarrows}} R \right)$$

A **2-signature** is a pair

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

1-signature

set of Σ-equations

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

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

Initial semantics for algebraic 2-signatures

Algebraic 2-signature: (Σ, E)

algebraic 1-signature

set of **elementary** Σ-equations

Theorem

Syntax exists for any algebraic 2-signature.

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

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

Initial semantics for algebraic 2-signatures

Algebraic 2-signature: (Σ, E)

algebraic 1-signature

set of **elementary** Σ-equations

Theorem

Syntax exists for any algebraic 2-signature.

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

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

Sketch of the construction of the syntax:

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

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

initial Σ-model morphism $\mathrm{i} : \mathrm{R} \to \mathrm{S}$

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:

$$\boldsymbol{\Sigma}_{\mathbf{LC\beta\eta}}(\mathrm{R}) := \Sigma_{\mathrm{LC}}(\mathrm{R}) = (\mathrm{R} \times \mathrm{R}) \coprod \mathrm{R}'$$

model of $\boldsymbol{\Sigma}_{\mathbf{LC}}$ = 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: $(\lambda \mathrm{x.t})\ \mathrm{u} = \underbrace{\mathrm{t}[\mathrm{x} \mapsto \mathrm{u}]}_{\sigma_{\mathrm{R}}(\mathrm{t},\mathrm{u})}$

η-equation: $\mathrm{t} = \lambda \mathrm{x}.(\mathrm{t}\ \mathrm{x})$

$$\mathbf{E}_{\mathbf{LC\beta\eta}} = \{\ \beta\text{-equation},\ \eta\text{-equation}\ \}$$

Example: λ-calculus modulo βη

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

$$\boldsymbol{\Sigma}_{\mathbf{LC\beta\eta}}(R) := \Sigma_{LC}(R) = (R \times R) \coprod R'$$

model of $\boldsymbol{\Sigma}_{\mathbf{LC}}$ = monad R with module morphisms:

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

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

$$\begin{tikzcd} R' \times R \arrow[rr, bend left, "\sigma_R"] \arrow[r, "\mathrm{abs} \times R"'] & R \times R \arrow[r, "\mathrm{app}"'] & R \end{tikzcd}$$

η-equation: $t = \lambda x.(t\ x)$

$$\begin{tikzcd} R \arrow[rr, bend left, "\mathrm{id}_R"] \arrow[r, "R\iota_1"'] & R' \arrow[r, "\mathrm{abs}"'] & R \end{tikzcd}$$

$$\mathbf{E}_{\mathbf{LC\beta\eta}} = \{\ \beta\text{-equation},\ \eta\text{-equation}\ \}$$

Example: fixpoint operator

Definition [AHLM CSL 2018]

A **fixpoint operator** in a monad R is a module morphism $\mathrm{fix} \colon \mathrm{R}' \to \mathrm{R}$

s.t. for any term $\mathrm{t} \in \mathrm{R}(\mathrm{X} \coprod \{\diamond\})$, $\mathrm{fix}(\mathrm{t}) = \mathrm{t}[\diamond \mapsto \mathrm{f}(\mathrm{t})]$

Intuition: $\mathtt{fix}(t) \;:=\;$ `let rec ◇ = t in ◇`

Algebraic 2-signature $(\Sigma_{\mathrm{fix}}, \mathrm{E}_{\mathrm{fix}})$ of a fixpoint operator:

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

$$E_{\mathtt{fix}} = \left\{ \begin{tikzcd} \begin{array}{c} R' \\ t \end{array} \arrow[r, bend left, "\mathtt{fix}(t)"] \arrow[r, bend right, "{t[\diamond \mapsto \mathtt{fix}(t)]}"'] & R \end{tikzcd} \right\}$$

Proposition [AHLM CSL 2018]

Fixpoint operators in $\mathrm{LC}_{\beta\eta}$ 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

$$(\Sigma_{\mathrm{fix}}, \mathrm{E}_{\mathrm{fix}}) \quad + \quad (\Sigma_{\mathrm{LC}\beta\eta}, \mathrm{E}_{\mathrm{LC}\beta\eta})$$

$$=$$

$$(\Sigma_{\mathrm{fix}} \coprod \Sigma_{\mathrm{LC}\beta\eta} \;,\; \mathrm{E}_{\mathrm{fix}} \cup \mathrm{E}_{\mathrm{LC}\beta\eta})$$

λ-calculus modulo βη 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:

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

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 $(\Sigma_{\text{mon}}, E_{\text{mon}})$ for the free commutative monoid monad:
$$\boldsymbol{\Sigma}_{\text{mon}}(\mathrm{R}) := 1 \coprod (\mathrm{R} \times \mathrm{R})$$

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

$$0 : 1 \to \mathrm{R} \qquad + : \mathrm{R} \times \mathrm{R} \to \mathrm{R}$$

4 elementary Σ-equations:

$$\begin{tikzcd} \underset{t}{R} \arrow[r, bend left, "0+t"] \arrow[r, bend right, "t"'] & R \end{tikzcd} \qquad \begin{tikzcd} \underset{s,t,u}{R \times R \times R} \arrow[r, bend left, "(s+t)+u"] \arrow[r, bend right, "s+(t+u)"'] & R \end{tikzcd}$$

$$\begin{tikzcd} \underset{t}{R} \arrow[r, bend left, "t+0"] \arrow[r, bend right, "t"'] & R \end{tikzcd} \qquad \begin{tikzcd} \underset{s,t}{R \times R} \arrow[r, bend left, "s+t"] \arrow[r, bend right, "t+s"'] & R \end{tikzcd}$$

Our target: dLC

Syntax of the *differential λ-calculus*:

Differential λ-terms

$$\begin{array}{rcll} s,t & ::= & \mathrm{x} & \\ & \mid & \lambda \mathrm{x}.\mathrm{t} & \multirow{2}{*}{$\left.\right\}$ λ-calculus} \\ & \mid & \mathrm{s}\ \mathrm{t} & \\ & \mid & \mathrm{D}\mathrm{s} \cdot \mathrm{t} & \\ & \mid & \mathrm{s} + \mathrm{t} & \multirow{2}{*}{$\left.\right\}$ free commutative monoid} \\ & \mid & 0 & \end{array}$$

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

$$\lambda \mathrm{x}.(\mathrm{s}+\mathrm{t}) = \lambda \mathrm{x}.\mathrm{s} + \lambda \mathrm{x}.\mathrm{t} \qquad \ldots$$

Algebraic 1-signature for dLC

Syntax of the *differential λ-calculus*:

Differential λ-terms

Corresponding 1-signature

$$
\begin{array}{rcll}
s,t & ::= & \mathrm{x} & \\
& | & \lambda \mathrm{x}.\mathrm{t} & \left.\right\} \\
& | & \mathrm{s}\ \mathrm{t} & \Sigma_{\mathrm{LC}}(\mathrm{R}) = \mathrm{R}' \coprod (\mathrm{R} \times \mathrm{R}) \\
& | & \mathrm{Ds} \cdot \mathrm{t} & \mathrm{R} \mapsto \mathrm{R} \times \mathrm{R} \\
& | & \mathrm{s} + \mathrm{t} & \left.\right\} \\
& | & 0 & \Sigma_{\mathrm{mon}}(\mathrm{R}) = 1 \coprod (\mathrm{R} \times \mathrm{R})
\end{array}
$$

Algebraic 1-signature for dLC

Syntax of the *differential λ-calculus*:

Differential λ-terms Corresponding 1-signature

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

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

$R \mapsto R \times R$

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

Resulting algebraic 1-signature: $\Sigma_{dLC}(R) = \Sigma_{LC}(R) \coprod (R \times R) \coprod \Sigma_{mon}(R)$

Elementary equations for dLC

Commutative monoidal structure:

$$
E_{\mathrm{mon}} \left\{ \begin{array}{c} s + t = t + s \\ s + (t + u) = (s + t) + u \\ 0 + t = t \\ t + 0 = t \end{array} \right.
\qquad
\begin{array}{r} R \times R \rightrightarrows R \\ R \times R \times R \rightrightarrows R \\ R \rightrightarrows R \\ R \rightrightarrows R \end{array}
$$

Linearity:

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

$\ldots$

n-ary fixpoint operator

Reminder: unary fixpoint operator in a monad R

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

Intuition: $\bar{t} \quad := \quad$ `let rec ◇ = t in ◇`

n-ary fixpoint operator:

$$\forall\, i \in \{1,..,\mathrm{n}\}, \quad \begin{array}{ccc} \mathrm{R}(\mathrm{X} \coprod \{\diamond_1, \ldots, \diamond_{\mathrm{n}}\})^{\mathrm{n}} & \rightarrow & \mathrm{R}(\mathrm{X}) \\ t_1, \ldots, t_n & \mapsto & \overline{t_i} \end{array} \qquad \textbf{s.t.} \quad \forall\, i,\ t_i \begin{bmatrix} \diamond_1 \mapsto \overline{t_1} \\ \cdots \\ \diamond_n \mapsto \overline{t_n} \end{bmatrix} = \overline{t_i}$$

Intuition:

$\overline{t_i} \quad := \quad$ `let rec` $\diamond_1$ `=` t_1 `and .. and` $\diamond_n$ `=` t_n `in` $\diamond_i$

n-ary fixpoint operator

n-ary fixpoint operator:

$$\forall\, i \in \{1,..,n\},$$

$$\begin{array}{ccc} \mathrm{R}(\mathrm{X} \coprod \{\diamond_1,\ldots,\diamond_n\})^{\mathrm{n}} & \rightarrow & \mathrm{R}(\mathrm{X}) \\ t_1,\ldots,t_n & \mapsto & \overline{t_i} \end{array}$$

s.t. $\forall\, i, \quad t_i \begin{bmatrix} \diamond_1 \mapsto \overline{t_1} \\ \cdots \\ \diamond_n \mapsto \overline{t_n} \end{bmatrix} = \overline{t_i}$

Algebraic 1-signature: $\Sigma_n(R) = \coprod_{i=1}^{n} (R^{\overbrace{\prime\cdots\prime}^{n}})^n$

n elementary equations $(\mathrm{R}^{\prime\cdots\prime})^{\mathrm{n}} \rightrightarrows \mathrm{R}$

$$\forall\, i, \qquad t_i \begin{bmatrix} \diamond_1 \mapsto \overline{t_1} \\ \cdots \\ \diamond_n \mapsto \overline{t_n} \end{bmatrix} = \overline{t_i}$$

Fixpoint operators

Syntax with fixpoint operators:

- for each n, a n-ary operator:

  $\texttt{let rec } \diamond_1 = t_1 \texttt{ and } .. \texttt{ and } \diamond_n = t_n \texttt{ in } \diamond_i$

- compatibility between these operators [AHLM CSL 2018]

Fixpoint operators

Syntax with fixpoint operators:

- for each n, a n-ary operator:

  $\texttt{let rec } \diamond_1 = t_1 \texttt{ and } .. \texttt{ and } \diamond_n = t_n \texttt{ in } \diamond_i$

- compatibility between these operators [AHLM CSL 2018]
  - invariance under **permutation**:

$$\begin{array}{l} \texttt{let rec } \diamond_1 = t_1 \\ \quad\quad \texttt{and } \diamond_2 = t_2 \\ \texttt{in } \diamond_1 \end{array} = \begin{array}{l} \texttt{let rec } \diamond_1 = t_2[\diamond_1 \leftrightarrow \diamond_2] \\ \quad\quad \texttt{and } \diamond_2 = t_1[\diamond_1 \leftrightarrow \diamond_2] \\ \texttt{in } \diamond_2 \end{array}$$

Fixpoint operators

Syntax with fixpoint operators:

- for each n, a n-ary operator:

  $\texttt{let rec } \diamond_1 = t_1 \texttt{ and } .. \texttt{ and } \diamond_n = t_n \texttt{ in } \diamond_i$

- compatibility between these operators [AHLM CSL 2018]
  - invariance under **permutation**:

$$\begin{array}{l} \texttt{let rec } \diamond_1 = t_1 \\ \quad\quad \texttt{and } \diamond_2 = t_2 \\ \texttt{in } \diamond_1 \end{array} \quad = \quad \begin{array}{l} \texttt{let rec } \diamond_1 = t_2[\diamond_1 \leftrightarrow \diamond_2] \\ \quad\quad \texttt{and } \diamond_2 = t_1[\diamond_1 \leftrightarrow \diamond_2] \\ \texttt{in } \diamond_2 \end{array}$$

  - invariance under **repetition**:

$$\begin{array}{l} \texttt{let rec } \diamond_1 = t \\ \quad\quad \texttt{and } \diamond_2 = t \\ \texttt{in } \diamond_1 \end{array} \quad = \quad \begin{array}{l} \texttt{let rec } \diamond_1 = t[\diamond_2 \mapsto \diamond_1] \\ \texttt{in } \diamond_1 \end{array}$$

Fixpoint operators

Syntax with fixpoint operators:

- for each n, a n-ary operator:

  $$\texttt{let rec } \diamond_1 = t_1 \texttt{ and } .. \texttt{ and } \diamond_n = t_n \texttt{ in } \diamond_i$$

- compatibility between these operators [AHLM CSL 2018]
  - invariance under **permutation**:

    $$\begin{array}{l} \texttt{let rec } \diamond_1 = t_1 \\ \quad\quad \texttt{and } \diamond_2 = t_2 \\ \texttt{in } \diamond_1 \end{array} \quad = \quad \begin{array}{l} \texttt{let rec } \diamond_1 = t_2[\diamond_1 \leftrightarrow \diamond_2] \\ \quad\quad \texttt{and } \diamond_2 = t_1[\diamond_1 \leftrightarrow \diamond_2] \\ \texttt{in } \diamond_2 \end{array}$$

  - invariance under **repetition**:

    $$\begin{array}{l} \texttt{let rec } \diamond_1 = t \\ \quad\quad \texttt{and } \diamond_2 = t \\ \texttt{in } \diamond_1 \end{array} \quad = \quad \begin{array}{l} \texttt{let rec } \diamond_1 = t[\diamond_2 \mapsto \diamond_1] \\ \texttt{in } \diamond_1 \end{array}$$

Fixpoint operators

Syntax with fixpoint operators:

- for each n, a n-ary operator:

  ```
  let rec ◇₁ = t₁ and .. and ◇ₙ = tₙ in ◇ᵢ
  ```

- compatibility between these operators [AHLM CSL 2018]

  In general:

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

  where $u : \{1, \ldots, p\} \to \{1, \ldots, q\}$

$$t_1, \ldots, t_q \in R(X \coprod \{\diamond_1, \ldots, \diamond_p\})$$

Fixpoint operators

Syntax with fixpoint operators:

- for each n, a n-ary operator:

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

- compatibility between these operators [AHLM CSL 2018]

  In general:

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

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

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

  $\Rightarrow$ Expressible as elementary equations $(\mathrm{R}^{,\dots,})^{\mathrm{q}} \rightrightarrows \mathrm{R}$.

Fixpoint operators

Syntax with fixpoint operators:

- for each n, a n-ary operator:

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

- compatibility between these operators [AHLM CSL 2018]

  In general:

`let rec` $\diamond_1 = t_1[\diamond_i \mapsto \diamond_{u(i)}]$ $\dots$ `and` $\diamond_q = t_q[\diamond_i \mapsto \diamond_{u(i)}]$ `in` $\diamond_{u(j)}$	=	`let rec` $\diamond_1 = t_{u(1)}$ $\dots$ `and` $\diamond_p = t_{u(p)}$ `in` $\diamond_j$

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

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

  $\Rightarrow$ Expressible as elementary equations $(\mathrm{R}^{, \dots ,})^{\mathrm{q}} \rightrightarrows \mathrm{R}$.

Fixpoint operators

Syntax with fixpoint operators:

- for each n, a n-ary operator:

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

- compatibility between these operators [AHLM CSL 2018]

  In general:

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

  where $u : \{1, \ldots, p\} \to \{1, \ldots, q\}$

$$t_1, \ldots, t_q \in R(X \coprod \{\diamond_1, \ldots, \diamond_p\})$$

  $\Rightarrow$ Expressible as elementary equations $(\mathrm{R}^{,\cdots,})^{\mathrm{q}} \rightrightarrows \mathrm{R}$.

Table of contents

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)

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 $\mathrm{s} : \Sigma(\mathrm{S}) \to \mathrm{S}$

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 $(\mathrm{S}\ , \mathrm{s})$

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 $(\mathrm{S}\ , \mathrm{s})$

2. Show that all the equations in E are satisfied for this model

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 $(\mathrm{S}\ , \mathrm{s})$

2. Show that all the equations in E are satisfied for this model

   $\Rightarrow$ induces a model of $(\Sigma\ , \mathrm{E})$

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 $(\mathrm{S}\,,\mathrm{s})$

2. Show that all the equations in E are satisfied for this model

   $\Rightarrow$ induces a model of $(\Sigma\,,\mathrm{E})$

Initiality of R $\Rightarrow$ model morphism $\mathrm{R} \to \mathrm{S}$ $\Rightarrow$ monad morphism $\mathrm{R} \to \mathrm{S}$

Example: Computing the set of free variables

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

$\mathcal{P}$ = power set monad

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

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

Example: Computing the set of free variables

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

$\mathcal{P}$ = power set monad

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

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

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

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

Example: Computing the set of free variables

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

$\mathcal{P}$ = power set monad

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

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

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

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

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

Example: Translating λ-calculus with fixpoint

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

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

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

λ-calculus modulo βη

monad morphism

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

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

a chosen fixpoint combinator

Example: Translating λ-calculus with fixpoint

$\mathrm{LC}_{\beta\eta\mathrm{fix}}$ = 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 $\mathrm{fix} : \mathrm{LC}_{\beta\eta\mathrm{fix}}' \to \mathrm{LC}_{\beta\eta\mathrm{fix}}$

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

λ-calculus modulo βη

monad morphism

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

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

a chosen fixpoint combinator

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

$\mathrm{app, abs}$

$$\hat{\mathrm{Y}} : \mathrm{LC}_{\beta\eta}' \to \mathrm{LC}_{\beta\eta}$$
$$\mathrm{t} \mapsto \mathrm{app(Y, abs(t))}$$

Example: Translating λ-calculus with fixpoint

$\mathrm{LC}_{\beta\eta\mathrm{fix}}$ = 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 $\mathrm{fix} : \mathrm{LC}_{\beta\eta\mathrm{fix}}' \to \mathrm{LC}_{\beta\eta\mathrm{fix}}$

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

λ-calculus modulo βη

monad morphism

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

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

a chosen fixpoint combinator

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

$\mathrm{app}, \mathrm{abs}$

$$\hat{\mathrm{Y}} : \mathrm{LC}_{\beta\eta}' \to \mathrm{LC}_{\beta\eta}$$
$$\mathrm{t} \mapsto \mathrm{app}(\mathrm{Y}, \mathrm{abs}(\mathrm{t}))$$

Initiality of $\mathrm{LC}_{\beta\eta\mathrm{fix}} \Rightarrow \mathrm{f} : \mathrm{LC}_{\beta\eta\mathrm{fix}} \to \mathrm{LC}_{\beta\eta}$

Example: Computing the size of a term

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

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

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

Example: Computing the size of a term

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

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

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

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

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

Example: Computing the size of a term

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

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

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

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

Solution [CSL AHLM 2018]:

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

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

Example: Computing the size of a term

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

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

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

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

Solution [CSL AHLM 2018]: continuation monad C(X) = $\mathbb{N}^{(\mathbb{N}^{X})}$

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

assigns an arbitrary size to each variable

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

Example: Computing the size of a term

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

Definition of a (~~monad~~) morphism $s : LC \to \mathbb{N}$ **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 2018]:

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

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

assigns an arbitrary size to each variable

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

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

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 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!