Higher-order Arities, Signatures and Equations via Modules

Ambroise Lafont

joint work with
Benedikt Ahrens, André Hirschowitz, Marco Maggesi
Keywords associated with syntax

Induction/Recursion

Substitution

Syntax

Operation/Construction

Model

Arity/Signature

This talk: give a discipline for specifying syntaxes
Motivating example: dLC

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

- explicitly involves **equations** e.g. $s+t = t+s$
- specifically 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
  ⇒ 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 $(\Delta_k)$. We set $\Delta_0 = \emptyset$ and $\Delta_{k+1}$ is defined as follows.

**Monotonicity:** if $t$ belongs to $\Delta_k$ then $t$ belongs to $\Delta_{k+1}$.

**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 $\Delta_{k+1}$. 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 $\sigma$ 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

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

belongs to $\Delta_{k+1}$. This term is identified with all the terms of the shape $D^u_1 \lambda x t \cdot (u_{\sigma(1)}, \ldots, u_{\sigma(n)}) \in \Delta_{k+1}$ where $\sigma$ is a permutation on $\{1, \ldots, n\}$.

**Application:** if $s \in \Delta_k$ and $t \in R(\Delta_k)$, then

$$(s)t$$

belongs to $\Delta_{k+1}$.

Setting $n = 0$ in the first two clauses, and restricting application by the constraint that $t \in \Delta_k \subseteq R(\Delta_k)$, 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 $\alpha$-conversion.
Let be given a denumerable set of variables. We define by induction on \( k \) an increasing family of sets \( (\Delta_k) \). We set \( \Delta_0 = \emptyset \) and \( \Delta_{k+1} \) is defined as follows.

**Monotonicity:** if \( t \) belongs to \( \Delta_k \) then \( t \) belongs to \( \Delta_{k+1} \).

**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 \( \Delta_{k+1} \). 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 \( \sigma \) 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

\[
D^n_1 \lambda x t \cdot (u_1, \ldots, u_n)
\]

belongs to \( \Delta_{k+1} \). This term is identified with all the terms of the shape \( D^n_1 \lambda x t \cdot (u_{\sigma(1)}, \ldots, u_{\sigma(n)}) \in \Delta_{k+1} \) where \( \sigma \) is a permutation on \( \{1, \ldots, n\} \).

**Application:** if \( s \in \Delta_k \) and \( t \in R(\Lambda_k) \), then

\[
(s)t \quad \text{as an operation: } \Lambda \times \text{FreeCommutativeMonoid}(\Lambda) \to \Lambda
\]

belongs to \( \Delta_{k+1} \).

Setting \( n = 0 \) in the first two clauses, and restricting application by the constraint that \( t \in \Delta_k \subseteq R(\Lambda_k) \), 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 \( \alpha \)-conversion.
Syntax of dLC: [BEM 2010]

A syntax for the differential $\lambda$-calculus by mutual induction:
[Bucciarelli-Ehrhard-Manzonetto 2010]

**Simple terms:**

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

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

$$\Lambda^d : T ::= 0 \mid s \mid s + T$$
Syntax of dLC: [BEM 2010]

A syntax for the **differential λ-calculus** by **mutual induction:**
[Bucciarelli-Ehrhard-Manzonetto 2010]

**Simple terms:**

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

**Differential λ-terms:**

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

- **variable**
- **modulo α-renaming of** \( x \)
- **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:**
\[ \Lambda^s : \quad s, t \quad ::= \quad x \mid \lambda x.s \mid sT \mid Ds \cdot t \]

**Differential λ-terms:**
\[ \Lambda^d : \quad T \quad ::= \quad 0 \mid s \mid s + T \]

\( \Lambda^d = \text{FreeCommutativeMonoid}(\Lambda^s) \)
Syntax of dLC: [BEM 2010]

A syntax for the differential $\lambda$-calculus by mutual induction:
[Bucciarelli-Ehrhard-Manzonetto 2010]

Simple terms:

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

Differential $\lambda$-terms:

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

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

Syntax: specified by operations and equations.
A syntax for the differential $\lambda$-calculus by mutual induction:
[Bucciarelli-Ehrhard-Manzonetto 2010]

**Simple terms:**

\[ \Lambda^s : \quad s, t \quad ::= \quad x \mid \lambda x. s \mid sT \mid D_s \cdot t \]

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

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

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

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 \(\lambda\)-terms:

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

\(\text{Differential } \lambda\text{-terms:} \)

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

neutral element for +

modulo commutativity and associativity

Macros in [BEM 2010]:

\[
\lambda x. \Sigma_i t_i := \Sigma_i \lambda x. t_i \\
(\Sigma_i t_i)u := \Sigma_i t_i u \\
D(\Sigma_i t_i) \cdot (\Sigma_j u_j) := \Sigma_i \Sigma_j Dt_i \cdot u_j
\]
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?

Syntax = operations + equations

Signature

Category of Models

Recursion

Substitution

generates a syntax = existence of the initial model
Table of contents

1. Signatures and models based on monads and modules

2. Equations

3. Recursion
1. 1-Signatures and models based on monads and modules
   • Substitution and monads
   • 1-Signatures and their models

2. Equations

3. Recursion
Example: differential $\lambda$-calculus

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

Free variable indexing:

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

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

$$dLC(\{x, y\}) = \{0, \lambda z. z, \ldots, x, y, x + y, \ldots\}$$
Example: differential λ-calculus

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

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:

\[t \mapsto t[x \mapsto f(x)]\]
Example: differential $\lambda$-calculus

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

Free variable indexing:

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

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

Parallel substitution:

\[
\text{bind}_f : dLC(X) \to dLC(Y) \quad \text{where} \quad f : X \to dLC(Y)
\]

\[
t \mapsto t[x \mapsto f(x)]
\]

\[
\Rightarrow (dLC, \text{var}_X : X \subset dLC(X), \text{bind}) = \text{monad on } \text{Set}
\]
Substitution and monads

**Example**: differential λ-calculus

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

**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**:

\[ \text{bind}_f : dLC(X) \rightarrow dLC(Y) \quad \text{where} \quad f : X \rightarrow dLC(Y) \]

\[ t \quad \mapsto \quad t[x \mapsto f(x)] \]

⇒ \((dLC, \text{var}_X : X \subset dLC(X), \text{bind}) = \textbf{monad on Set}\)

**Monad morphism** = mapping preserving variables and substitutions.
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

+ commutes with substitution

\(dLC \times dLC\) is a module over \(dLC\)

\(+: dLC \times dLC \rightarrow dLC\) is a module morphism
Building blocks for specifying operations

Essential constructions of modules over a monad $R$:

- $R$ itself

- $M \times N$ for any modules $M$ and $N$
  
  e.g. $R \times R$:  
  
  $f : X \rightarrow R(Y)$

  $(t,u)[x \mapsto f(x)] := (t[x \mapsto f(x)], u[x \mapsto f(x)])$

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

  used to model an operation binding a variable (Cf next slide).
operations = module morphisms = maps commuting with substitution.

\[
\begin{align*}
0 &: \ 1 \to dLC \\
 &\text{app} : dLC \times dLC \to dLC \\
+ &: dLC \times dLC \to dLC \\
 &\text{abs} : dLC' \to dLC \\
\text{abs}_X &: dLC(X \coprod \{\Box\}) \to dLC(X) \\
\end{align*}
\]

\[
t \mapsto \lambda\Box.t
\]
Syntactic operations are module morphisms

\[
\text{operations} = \text{module morphisms} = \text{maps commuting with substitution.}
\]

\[
\begin{align*}
0 & : 1 \rightarrow \text{dLC} & \text{app} & : \text{dLC} \times \text{dLC} \rightarrow \text{dLC} \\
+ & : \text{dLC} \times \text{dLC} \rightarrow \text{dLC} & \text{abs} & : \text{dLC}' \rightarrow \text{dLC} \\
\end{align*}
\]

\[
\text{abs}_x : \text{dLC}(X \sqcup \{\diamond\}) \rightarrow \text{dLC}(X) \\
\quad t \mapsto \lambda\diamond.t
\]

Combining operations into a single one using disjoint union

\[
\begin{align*}
[0, +] & : 1 \sqcup (\text{dLC} \times \text{dLC}) \rightarrow \text{dLC} \\
[\text{app}, \text{abs}] & : (\text{dLC} \times \text{dLC}) \sqcup \text{dLC}' \rightarrow \text{dLC}
\end{align*}
\]
Syntactic operations are module morphisms

operations = module morphisms = maps commuting with substitution.

\[ 0 : \ 1 \rightarrow dLC \]

\[ \text{app} : dLC \times dLC \rightarrow dLC \]

\[ + : dLC \times dLC \rightarrow dLC \]

\[ \text{abs} : dLC' \rightarrow dLC \]

\[ \text{abs}_x : dLC(X \coprod \{\diamond\}) \rightarrow dLC(X) \]

\[ t \mapsto \lambda\diamond.t \]

Combining operations into a single one using disjoint union

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

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

\[ [\text{app}, \text{abs}, 0, +] : (dLC \times dLC) \coprod dLC' \coprod 1 \coprod (dLC \times dLC) \rightarrow dLC \]
1-signatures and their models

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

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

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

$$\left( R, \rho : \Sigma(R) \to R \right)$$

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

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

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

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

A **model morphism** $m : (R, \rho) \to (S, \sigma) = \text{monad morphism commuting}$

with the module morphism:

$$\Sigma(m) \quad \rho \quad R$$

$$\Sigma(S) \quad \sigma \quad S$$

$$R$$

$$m$$
**1-signatures and their models**

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

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

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

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

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

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

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

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

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

$$\Sigma(R) \xrightarrow{\rho} R$$

$$\Sigma(m) \downarrow \quad m \downarrow$$

$$\Sigma(S) \xrightarrow{\sigma} S$$
A **1-signature** $\Sigma = \text{functorial assignment}:

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

**monad module over** $R$

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

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

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

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

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

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

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

\[
\begin{align*}
\Sigma(R) & \xrightarrow{\rho} R \\
\Sigma(m) & \downarrow \\
\Sigma(S) & \xrightarrow{\sigma} S \\
\end{align*}
\]

\[
\begin{align*}
\Sigma(R) & \xrightarrow{\rho} R \\
\downarrow & \\
\Sigma(S) & \xrightarrow{\sigma} S \\
\downarrow & \\
m & \\
\end{align*}
\]
A **1-signature** $\Sigma$ is a functorial assignment:

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

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

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

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

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

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

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

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

$$[0, +] : 1 \coprod (dLC \times dLC) \to dLC$$
1-signatures and their models

A **1-signature** \( \Sigma = \text{functorial assignment} \):
\[
R \mapsto \Sigma(R)
\]

monad
module over \( \mathbb{R} \)

A **model of** \( \Sigma \) is a pair:
\[
(R, \quad \rho : \Sigma(R) \to R)
\]

monad
module morphism

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

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

**Example**: \((0,+))
\[
\Sigma_{0,+}(R) = 1 \coprod (R \times R)
\]

\(dLC = \text{model of } \Sigma_{0,+} \)
\[
[0, +] : 1 \coprod (dLC \times dLC) \to dLC
\]
Definition

Given a 1-signature $\Sigma$, its syntax is an initial object in its category of models.

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

Answer: No.
Given a 1-signature $\Sigma$, its syntax is an initial object in its category of models.

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

**Answer**: No.

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

powerset endofunctor on Set
Examples of 1-signatures generating syntax

- **(0,+)** language:
  - Signature: \( R \mapsto 1 \amalg (R \times R) \)
  - Model: \((R, \ 0 : 1 \to R, \ + : R \times R \to R)\)
  - Syntax: \((B, \ 0 : 1 \to B, \ + : B \times B \to B)\)

- **lambda calculus**:
  - Signature: \( R \mapsto R' \amalg (R \times R) \)
  - Model: \((R, \ abs : R' \to R, \ app : R \times R \to R)\)
  - Syntax: \((\Lambda, \ abs : \Lambda' \to \Lambda, \ app : \Lambda \times \Lambda \to \Lambda)\)

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 signature = 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 $\lambda$-calculus.
Quotients of algebraic 1-signatures

[AHLM CSL 2018]: notion of *quotients* of 1-signatures.

**Theorem [AHLM CSL 2018]**

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

**Examples:**

- a *commutative* binary operation
- application of the differential λ-calculus (original variant)
  
  \[\text{app} : \text{dLC} \times \text{FreeCommutativeMonoid(dLC)} \rightarrow \text{dLC}\]
Quotients of algebraic 1-signatures

[AHLM CSL 2018]: notion of \textit{quotients} of 1-signatures.

\begin{itemize}
  \item associativity of \(\text{+}\)
  \item linearity of the operations
\end{itemize}

\begin{quote}
\textbf{Theorem [AHLM CSL 2018]} \hspace{1cm}
Syntax exists for any \textit{quotient} of algebraic 1-signature.
\end{quote}

\textbf{Examples:}

\begin{itemize}
  \item a \textbf{commutative} binary operation
  \item application of the differential \(\lambda\)-calculus (original variant)
    
    \hspace{1cm} \textbf{app} : \textit{dLC} \times \textit{FreeCommutativeMonoid(dLC)} \to \textit{dLC}
\end{itemize}

... but not enough for the differential \(\lambda\)-calculus:

\begin{itemize}
  \item \textbf{associativity} of \(\text{+}\)
  \item \textbf{linearity} of the operations
\end{itemize}
1. 1-Signatures and models based on monads and modules

2. Equations

3. Recursion
Example: a commutative binary operation

Specification of a binary operation

1-Signature: \( R \mapsto R \times R \)

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

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

Specification of a **commutative** binary operation

1-Signature: \( R \mapsto R \times R \)
Model: \((R, \ + : R \times R \to R) \) \quad s.t. \quad t + u = u + t \quad (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:

\[
\begin{array}{ccc}
R \times R & \xrightarrow{\text{swap}} & R \times R \\
& \text{+} & \\
& \text{+} & \\
& \xrightarrow{\text{+}} & R
\end{array}
\]
Given a 1-signature $\Sigma$, (e.g. binary operation: $\Sigma(R) = R \times R$)

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

$$R \mapsto \left( \begin{array}{c}
A(R) \rightarrowtail B(R)
\end{array} \right)$$

model of $\Sigma$

parallel pair of module morphisms over $R$

A 2-signature is a pair

$$(\Sigma, E)$$

1-signature set of $\Sigma$-equations

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

- a model $R$ of $\Sigma$
- s.t. $\forall (A \Rightarrow B) \in E$, the two morphisms $A(R) \Rightarrow B(R)$ are equal
Initial semantics for algebraic 2-signatures

**Algebraic 2-signature:**

\[(\Sigma, E)\]

**algebraic 1-signature**

set of **elementary** \(\Sigma\)-equations

**Theorem**

Syntax exists for any algebraic 2-signature.

Main instances of **elementary** \(\Sigma\)-equations \(A \Rightarrow B\):

- \(A = \text{algebraic 1-signature}\) e.g. \(A(R) = R \times R\)
- \(B(R) = R\)
Initial semantics for algebraic 2-signatures

**Algebraic 2-signature:**

$$(\Sigma, E)$$

**algebraic 1-signature**

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

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

**Theorem**

Syntax exists for any algebraic 2-signature.

**Sketch of the construction of the syntax:**

Quotient the initial model $R$ of $\Sigma$ by the following relation:

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

initial $\Sigma$-model morphism $i : R \to S$
Example: $\lambda$-calculus modulo $\beta\eta$

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

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

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

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

**$\beta$-equation:** $(\lambda x.t) u = t[x \mapsto u]$

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

$$E_{\text{LC}\beta\eta} = \{ \beta\text{-equation}, \eta\text{-equation} \}$$
Example: $\lambda$-calculus modulo $\beta\eta$

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

$$\Sigma_{LC\beta\eta}(R) := \Sigma_{LC}(R) = (R \times R) \sqcup R'$$

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

- $\text{app} : R \times R \to R$
- $\text{abs} : R' \to R$

**β-equation:** $(\lambda x.t) \ u = t[x \mapsto u]

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

$$E_{LC\beta\eta} = \{ \text{β-equation, η-equation} \}$$
Example: fixpoint operator

Definition [AHLM CSL 2018]

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

**Intuition:**

- $\text{fix}(t) := \text{let rec } \diamond = t \text{ in } \diamond$
- [AHLM CSL 2018] Fixpoint operator in $\text{LC}_{\beta \eta} \simeq \text{fixpoint combinators}$
A **fixpoint operator** in a monad $R$ is a module morphism $\text{fix}: R' \to R$ s.t. for any term $t \in R(X \coprod \{\diamond\})$, $\text{fix}(t) = t[\diamond \mapsto f(t)]$

**Intuition:**

- $\text{fix}(t) := \texttt{let rec } \diamond = t \texttt{ in } \diamond$
- [AHLM CSL 2018] Fixpoint operator in $\text{LC}_{\beta\eta} \simeq$ fixpoint combinators

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

$$\Sigma_{\text{fix}}(R) := R' \quad E_{\text{fix}} = \begin{cases} \text{fix}(t) \quad R' \to R \\ t \quad R \to R' \\ t[\diamond \mapsto \text{fix}(t)] \end{cases}$$
Combining algebraic 2-signatures

Algebraic 2-signatures can be combined:

\[ (\Sigma_{\text{fix}}, E_{\text{fix}}) \oplus (\Sigma_{\text{LC} \beta \eta}, E_{\text{LC} \beta \eta}) = (\Sigma_{\text{fix}} \uplus \Sigma_{\text{LC} \beta \eta}, E_{\text{fix}} \cup E_{\text{LC} \beta \eta}) \]

\[ \lambda\text{-calculus modulo } \beta \eta \text{ with an explicit fixpoint operator} \]
Example: free commutative monoid

Algebraic 2-signature \((\Sigma_{\text{mon}}, E_{\text{mon}})\) for the free commutative monoid monad:

\[
\Sigma_{\text{mon}}(R) := 1 \coprod (R \times R)
\]

**model of** \(\Sigma_{\text{mon}} = \text{monad } R\) with module morphisms:

\[
0 : 1 \to R \quad + : R \times R \to R
\]
Example: free commutative monoid

Algebraic 2-signature \((\Sigma_{\text{mon}}, E_{\text{mon}})\) for the free commutative monoid monad:

\[
\Sigma_{\text{mon}}(R) := 1 \coprod (R \times R)
\]

**model of** \(\Sigma_{\text{mon}} = \text{monad } R\) with module morphisms:

\[
0 : 1 \to R \quad + : R \times R \to R
\]

3 elementary \(\Sigma\)-equations:

\[
\begin{align*}
R \times R \times R & \xrightarrow{(s+t)+u} R \\
R \times R & \xrightarrow{s+(t+u)} R \\
R \times R & \xrightarrow{s+t} R \\
R & \xrightarrow{0+t} R \\
R & \xrightarrow{t+s} R
\end{align*}
\]
Our target: dLC

Syntax of the differential $\lambda$-calculus:

Differential $\lambda$-terms

\[
s, t ::= x \\
\quad | \lambda x.t \\
\quad | st \\
\quad | Ds \cdot t \\
\quad | s + t \\
\quad | 0
\]

\[
\} \quad \lambda$-calculus
\]

\[
\} \quad \text{free commutative monoid}
\]

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

\[
\lambda x. (s + t) = \lambda x.s + \lambda x.t
\]

\[
\ldots
\]
Syntax of the differential λ-calculus:

**Differential λ-terms**

\[
s, t ::= x \\
\quad | \lambda x . t \\
\quad | s \cdot t \\
\quad | Ds \cdot t \\
\quad | s + t \\
\quad | 0
\]

(variables \(\subset R\) for any monad \(R\))

**Corresponding 1-signature**

\[
\Sigma_{LC}(R) = R' \uplus (R \times R)
\]

\[
\Sigma_{\text{mon}}(R) = 1 \uplus (R \times R)
\]
Algebraic 1-signature for dLC

Syntax of the differential $\lambda$-calculus:

**Differential $\lambda$-terms**

\[
s, t ::= x \\
| \lambda x.t \\
| s \cdot t \\
| Ds \cdot t \\
| s + t \\
| 0
\]

**Corresponding 1-signature**

\[
\Sigma_{\text{LC}}(R) = R' \coprod (R \times R)
\]

\[
R \mapsto R \times R
\]

\[
\Sigma_{\text{mon}}(R) = 1 \coprod (R \times R)
\]

Resulting algebraic 1-signature:

\[
\Sigma_{\text{dLC}}(R) = \Sigma_{\text{LC}}(R) \coprod (R \times R) \coprod \Sigma_{\text{mon}}(R)
\]
Elementary equations for dLC

Commutative monoidal structure:

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

Linearity:

\[ \lambda x. (s + t) = \lambda x.s + \lambda x.t & R \times R \implies R \\
 D(s + t) \cdot u = Ds \cdot u + Dt \cdot u & R \times R \times R \implies R \\
 Ds \cdot (t + u) = Ds \cdot t + Ds \cdot u & R \times R \times R \implies R \]

\ldots \]
n-ary fixpoint operator

Reminder: unary fixpoint operator in a monad \( R \)

\[
R(X \coprod \{\diamond\}) \quad \mapsto \quad R(X) \quad \text{s.t.} \quad t[\diamond \mapsto \overline{t}] = \overline{t}
\]

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

n-ary fixpoint operator:

\[
\forall i \in \{1, \ldots, n\}, \quad R(X \coprod \{\diamond_1, \ldots, \diamond_n\})^n \quad \mapsto \quad R(X) \quad \text{s.t.} \quad \forall i, t_i \left[ \begin{array}{c} \diamond_1 \mapsto \overline{t_1} \\ \vdots \\ \diamond_n \mapsto \overline{t_n} \end{array} \right] = \overline{t_i}
\]

Intuition: \( \overline{t_i} := \text{let rec } \diamond_1 = t_1 \text{ and } \ldots \text{ and } \diamond_n = t_n \text{ in } \diamond_i \)
Reminder: unary fixpoint operator in a monad $R$

$$R(X \sqcup \{\diamond\}) \to R(X) \quad \text{s.t.} \quad t[\diamond \mapsto \bar{t}] = \bar{t}$$

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

**n-ary fixpoint operator:**

$$\forall i \in \{1,..,n\}, \quad R(X \sqcup \{\diamond_1,\ldots,\diamond_n\})^n \to R(X) \quad \text{s.t.} \quad \forall i, t_i \quad \begin{bmatrix} \diamond_1 \mapsto \bar{t}_1 \\ \cdots \\ \diamond_n \mapsto \bar{t}_n \end{bmatrix} = \bar{t}_i$$

Intuition: \(\bar{t}_i := \text{let rec } \diamond_1 = t_1 \text{ and } \ldots \text{ and } \diamond_n = t_n \text{ in } \diamond_i\)

\(\Rightarrow\) specifiable as an algebraic 2-signature
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]
Syntax with fixpoint operators:

• for each n, a n-ary operator:
  \[
  \text{let rec } \diamond_1 = t_1 \text{ and } \ldots \text{ and } \diamond_n = t_n \text{ in } \diamond_i
  \]

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

  \[
  \text{let rec } \diamond_1 = t_1 \text{ and } \diamond_2 = t_2 \text{ in } \diamond_1 \equiv \text{let rec } \diamond_1 = t_2[\diamond_1 \leftrightarrow \diamond_2] \text{ and } \diamond_2 = t_1[\diamond_1 \leftrightarrow \diamond_2] \text{ in } \diamond_2
  \]
Fixpoint operators

Syntax with fixpoint operators:

- for each $n$, a $n$-ary operator:

$$
\text{let rec } \diamond_1 = t_1 \text{ and } \ldots \text{ and } \diamond_n = t_n \text{ in } \diamond_i
$$

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

    $$
    \begin{align*}
    \text{let rec } \diamond_1 &= t_1 \\
    \text{and } \diamond_2 &= t_2 \\
    \text{in } \diamond_1
    \end{align*}
    \quad \Rightarrow \quad 
    \begin{align*}
    \text{let rec } \diamond_1 &= t_2[\diamond_1 \leftrightarrow \diamond_2] \\
    \text{and } \diamond_2 &= t_1[\diamond_1 \leftrightarrow \diamond_2] \\
    \text{in } \diamond_2
    \end{align*}
    $$

  - invariance under repetition:

    $$
    \begin{align*}
    \text{let rec } \diamond_1 &= t \\
    \text{and } \diamond_2 &= t \\
    \text{in } \diamond_1
    \end{align*}
    \quad \Rightarrow \quad 
    \begin{align*}
    \text{let rec } \diamond_1 &= t[\diamond_2 \leftrightarrow \diamond_1] \\
    \text{in } \diamond_1
    \end{align*}
    $$
Syntax with fixpoint operators:

• for each n, a n-ary operator:

\[
\text{let rec } \diamond_1 = t_1 \text{ and } \ldots \text{ and } \diamond_n = t_n \text{ in } \diamond_i
\]

• compatibility between these operators [AHLM CSL 2018]

**general form:**

\[
\begin{align*}
\text{let rec } \diamond_1 &= t_1[\diamond_i \mapsto \diamond_{u(i)}] \\
&\quad \ldots \\
\text{and } \diamond_q &= t_q[\diamond_i \mapsto \diamond_{u(i)}] \\
\text{in } \diamond_{u(j)}
\end{align*}
\]

\[
\begin{align*}
\text{let rec } \diamond_1 &= t_{u(1)} \\
&\quad \ldots \\
\text{and } \diamond_p &= t_{u(p)} \\
\text{in } \diamond_j
\end{align*}
\]

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

\[ t_1, \ldots, t_q \in R(X \bigsqcup \{\diamond_1, \ldots, \diamond_p\}) \]
Fixpoint operators

Syntax with fixpoint operators:

• for each n, a n-ary operator:
  
  \[
  \text{let rec } \diamond_1 = t_1 \text{ and .. and } \diamond_n = t_n \text{ in } \diamond_i
  \]

• compatibility between these operators [AHLM CSL 2018]

  general form:

  \[
  \begin{align*}
  \text{let rec } \diamond_1 &= t_1[\diamond_i \mapsto \diamond_{u(i)}] \\
  & \quad \ldots \\
  & \quad \text{and } \diamond_q = t_q[\diamond_i \mapsto \diamond_{u(i)}] \\
  \text{in } \diamond_{u(j)}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{let rec } \diamond_1 &= t_{u(1)} \\
  & \quad \ldots \\
  & \quad \text{and } \diamond_p = t_{u(p)} \\
  \text{in } \diamond_{j}
  \end{align*}
  \]

  where \( u : \{1, \ldots, p\} \rightarrow \{1, \ldots, q\} \)
  \( t_1, \ldots, t_q \in R(X \coprod \{\diamond_1, \ldots, \diamond_p\}) \)
  \( \Rightarrow \) Expressible as elementary equations \((R^\ldots)^q \Rightarrow R\).
Table of contents

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

2. Equations

3. Recursion
Principle of recursion

Recursion on the syntax = Initiality in the category of models

Recipe for constructing "by recursion" a monad morphism:

\[ f : R \to S \]

initial model of a 2-signature \((\Sigma, E)\)
Principle of recursion

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

**Recipe for constructing "by recursion" a monad morphism:**

\[ f : R \rightarrow S \]

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

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

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

**Recipe for constructing "by recursion" a monad morphism:**

\[
\begin{aligned}
&f : R \rightarrow S \\
\text{initial model of a 2-signature } (\Sigma, E)
\end{aligned}
\]

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

Recursion on the syntax ≈ Initiality in the category of models

Recipe for constructing "by recursion" a monad morphism:

\[ f : R \rightarrow S \]

initial model of a 2-signature \((\Sigma, E)\)

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

2. Show that all the equations in \(E\) are satisfied for this model
Principle of recursion

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

**Recipe for constructing "by recursion" a monad morphism:**

1. Give a module morphism $s : \Sigma(S) \to S$
2. Show that all the equations in $E$ are satisfied for this model

\[ f : R \to S \]

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

$\Rightarrow$ induces a $\Sigma$-model $(S, s)$

$\Rightarrow$ induces a model of $(\Sigma, E)$
Principle of recursion

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

Recipe for constructing "by recursion" a monad morphism:

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

2. Show that all the equations in $E$ are satisfied for this model
   $\Rightarrow$ induces a model of $(\Sigma, E)$

Initiality of $R$ $\Rightarrow$ model morphism $R \rightarrow S$ $\Rightarrow$ monad morphism $R \rightarrow S$
Example: Computing the set of free variables

\[ \lambda \text{-calculus monad} \]

\[ \text{fv}_X : \text{LC}(X) \rightarrow \mathcal{P}(X) \]

\[ t \mapsto \text{(exact) set of free variables of } t \]
Example: Computing the set of free variables

\( \lambda \)-calculus monad

\[ \text{fv}_X : \text{LC}(X) \rightarrow \mathcal{P}(X) \]

\( t \mapsto (\text{exact}) \text{ set of free variables of } t \)

.. as a monad morphism \( \text{fv} : \text{LC} \rightarrow \mathcal{P} \)

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

\( \Rightarrow \) make \( \mathcal{P} \) a model of \( \Sigma_{\text{LC}} \)

\[ \cup : \mathcal{P} \times \mathcal{P} \rightarrow \mathcal{P} \]

\( \Sigma_{\text{LC}}(R) = (R \times R) \bigcup R' \)

\( -\{ \diamond \} : \mathcal{P}' \rightarrow \mathcal{P} \)
Example: Computing the set of free variables

\[ \lambda \text{-calculus monad} \]

\[ \text{fv}_X : \text{LC}(X) \to P(X) \]

\[ t \mapsto \text{(exact) set of free variables of } t \]

.. as a monad morphism \( \text{fv} : \text{LC} \to P \)

\[ \Sigma_{\text{LC}}(R) = (R \times R) \coprod R' \]

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

\( \Rightarrow \) make \( P \) a model of \( \Sigma_{\text{LC}} \)

\[ \cup : P \times P \to P \]

\[ _{\setminus}\{ \diamond \} : P' \to P \]

Initiality of \( \text{LC} \) \( \Rightarrow \text{fv} : \text{LC} \to P \)
Example: Computing the set of free variables

\[ \lambda \text{-calculus monad} \]

\[ \text{fv}_X : \text{LC}(X) \rightarrow \mathcal{P}(X) \]

\[ t \mapsto (\text{exact}) \text{ set of free variables of } t \]

.. as a monad morphism \( \text{fv} : \text{LC} \rightarrow \mathcal{P} \)

\[ \text{LC} = \text{initial model of } (\Sigma_{\text{LC}}, \emptyset) \]

\[ \Sigma_{\text{LC}}(R) = (R \times R) \bigsqcup R' \]

\[ \Rightarrow \text{make } \mathcal{P} \text{ a model of } \Sigma_{\text{LC}} \]

\[ \cup : \mathcal{P} \times \mathcal{P} \rightarrow \mathcal{P} \]

\[ \setminus\{\Diamond\} : \mathcal{P}' \rightarrow \mathcal{P} \]

Initiality of \( \text{LC} \) \( \Rightarrow \) \( \text{fv} : \text{LC} \rightarrow \mathcal{P} \)

Equalities as a monad morphism:

\[ \text{fv}(x) = \{x\} \]

\[ \text{fv}(t[x \mapsto u(x)]) = \bigcup_{x \in \text{fv}(t)} \text{fv}(u(x)) \]

Equalities as a model morphism:

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

\[ \text{fv}(\text{abs}(t)) = \text{fv}(t) \setminus \{\Diamond\} \]
Example: Translating $\lambda$-calculus with fixpoint

$\lambda$-calculus modulo $\beta\eta$ + fixpoint operator $\text{fix}$

\[ \text{compilation} \quad \mapsto \quad \lambda$-calculus modulo $\beta\eta \]

$\text{fix}(t) \mapsto ?$
Example: Translating $\lambda$-calculus with fixpoint

$\lambda$-calculus modulo $\beta\eta$ + fixpoint operator $\text{fix}$

\[
\begin{align*}
\text{compilation} & : \quad \lambda\text{-calculus modulo } \beta\eta \\
\text{fix}(t) & \mapsto ?
\end{align*}
\]

...as a monad morphism $\text{LC}_{\beta\eta + \text{fix}} \to \text{LC}_{\beta\eta}$

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

$\Rightarrow$ make $\text{LC}_{\beta\eta}$ a model of $(\Sigma_{\text{LC}_{\beta\eta}}, E_{\text{LC}_{\beta\eta}}) + (\Sigma_{\text{fix}}, E_{\text{fix}})$:
Example: Translating $\lambda$-calculus with fixpoint

$\lambda$-calculus modulo $\beta\eta$
+ fixpoint operator $\text{fix}$

$\text{compilation}$

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

$\text{fix}(t) \mapsto ?$

...as a monad morphism

$LC_{\beta\eta+\text{fix}} \to LC_{\beta\eta}$

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

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

app, abs

a fixpoint operator in $LC_{\beta\eta}$
Example: Translating $\lambda$-calculus with fixpoint

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

+ fixpoint operator $\text{fix}$

\[
\begin{align*}
\text{compilation} & \quad \longrightarrow \\
\text{fix}(t) & \quad \mapsto \quad ?
\end{align*}
\]

...as a monad morphism

\[
\begin{align*}
\text{LC}_{\beta\eta+\text{fix}} & \quad \rightarrow \\
\text{LC}_{\beta\eta}
\end{align*}
\]

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

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

- app, abs
- a fixpoint operator in $\text{LC}_{\beta\eta}$

**Proposition [AHLM CSL 2018]**

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

\( \lambda \)-calculus modulo \( \beta \eta \) + fixpoint operator \( \text{fix} \)

\[
\text{compilation} \quad \quad \quad \quad \lambda \text{-calculus modulo } \beta \eta \\
\text{fix}(t) \mapsto \text{app}(Y, \text{abs}(t))
\]

...as a monad morphism

\[
\mathbf{LC}_{\beta \eta + \text{fix}} \rightarrow \mathbf{LC}_{\beta \eta}
\]

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

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

\[
\text{app, abs}
\]

\[
\hat{\text{Y}} : t \mapsto \text{app}(Y, \text{abs}(t))
\]

Proposition [AHLM CSL 2018]

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

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

+ fixpoint operator fix

\[ \text{compilation} \quad \overset{\Rightarrow}{\longrightarrow} \quad \lambda$-calculus modulo $\beta\eta \]

\[ \text{fix}(t) \mapsto \text{app}(Y, \text{abs}(t)) \]

...as a monad morphism

\[ \text{LC}_{\beta\eta+fix} \rightarrow \text{LC}_{\beta\eta} \]

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

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

\[ \text{app, abs} \]

\[ \text{a fixpoint operator in } \text{LC}_{\beta\eta} \]

\[ \hat{Y} : t \mapsto \text{app}(Y, \text{abs}(t)) \]

Proposition [AHLM CSL 2018]

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

Initiability of $\text{LC}_{\beta\eta+fix} \Rightarrow$ monad morphism $\text{LC}_{\beta\eta+fix} \rightarrow \text{LC}_{\beta\eta}$
Example: Computing the size of a term

\( \lambda \)-calculus monad

\[
\begin{align*}
\mathsf{s}_x & : \mathsf{LC}(X) \to \mathbb{N} \\
t & \mapsto \text{number of constructors in } t
\end{align*}
\]

.. as a monad morphism \( s : \mathsf{LC} \to \mathbb{N} \)

\[
\begin{align*}
\mathsf{s}(x) &= 0 \\
\mathsf{s}(\lambda x.x) &= 1 \\
\mathsf{s}((\lambda x.x) \ y) &= 2
\end{align*}
\]
Example: Computing the size of a term

\begin{itemize}
  \item \( s(x) = 0 \)
  \item \( s(\lambda x. x) = 1 \)
  \item \( s((\lambda x. x) \ y) = 2 \)
\end{itemize}

\( s_X : \text{LC}(X) \rightarrow \mathbb{N} \)

\( \text{number of constructors in } t \)

\( \textbf{\text{\textcolor{red}{as a monad morphism}} } s : \text{LC} \rightarrow \mathbb{N} \)

\( \mathbb{N} \text{ is not a monad} ! \)
Example: Computing the size of a term

\( \lambda \)-calculus monad

\[
s_x : \text{LC}(X) \to \mathbb{N} \\
\text{number of constructors in } t
\]

.. as a monad morphism \( s : \text{LC} \to \mathbb{N} \)

\( \mathbb{N} \) is not a monad!

Solution [CSL AHLM 2018]:

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

continuation monad \( \text{C}(X) = \mathbb{N}^{(\mathbb{N}^X)} \)
Example: Computing the size of a term

**λ-calculus monad**

\[ s_X : \text{LC}(X) \rightarrow \mathbb{N} \]

\[ t \mapsto \text{number of constructors in } t \]

**as a monad morphism** \( s : \text{LC} \rightarrow \mathbb{N} \)

\( \mathbb{N} \) is not a monad!

**Solution** [CSL AHLM 2018]:

1. define \( f : \text{LC} \rightarrow C \) by recursion
2. deduce \( s : \text{LC} \rightarrow \mathbb{N} \)

**Intuition:** \( f_X : \text{LC}(X) \rightarrow \mathbb{N}^{(\mathbb{N}^X)} \)

\[ \text{uncurry} \]

\( g : \text{LC}(X) \times \mathbb{N}^X \rightarrow \mathbb{N} \)

\[ g(x, u) = u(x) \]
Example: Computing the size of a term

\( \lambda \text{-calculus monad} \)

\[
\begin{align*}
  s_X & : \text{LC}(X) \to \mathbb{N} \\
  t & \mapsto \text{number of constructors in } t
\end{align*}
\]

\( \text{.. as a monad morphism } s : \text{LC} \to \mathbb{N} \)

\( \mathbb{N} \text{ is not a monad!} \)

**Solution** [CSL AHLM 2018]:

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

**Intuition:** \( f_X : \text{LC}(X) \to \mathbb{N}^{(\mathbb{N}^X)} \) \( \text{uncurry} \Rightarrow g : \text{LC}(X) \times \mathbb{N}^X \to \mathbb{N} \)

\[
g(x, u) = u(x)
\]

\[
s(t) = g(t, (x \mapsto 0))
\]

\( \text{variables are of size 0} \)
Conclusion

**Summary of the talk:**

- notion of 1-signature and models based on monads and modules
- 2-signature = 1-signature + set of equations
- *algebraic* 2-signatures generate a syntax, e.g. differential λ-calculus.

Main theorems 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:

- notion of 1-signature and models based on monads and modules
- 2-signature = 1-signature + set of equations
- algebraic 2-signatures generate a syntax, e.g. differential λ-calculus.

Main theorems formalized in Coq using the UniMath library.

Future work:

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

Thank you!