Signatures and models for syntax and operational semantics in the presence of variable binding

Ambroise Lafont[1]

[1]DAPI
IMT Atlantique

PhD supervised by N. Tabareau and T. Hirschowitz, 2019

Motivation

How do we formally specify a **programming language**?

In the literature: no common well-established discipline.

Differential λ-calculus [Ehrhard-Regnier 2003]

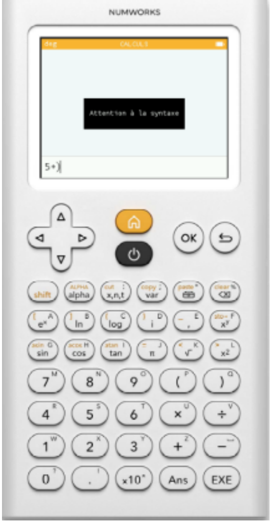
$\sim$10 pages (section 2 $\rightarrow$ beginning of section 3) describing the programming language and proving some properties.

This PhD: a discipline for presenting programming languages

- from small elementary data
- automatically ensuring some properties

What is a programming language?

Example: arithmetic expressions in a calculator



Syntax (of expressions) = formal language

- *vocabulary* : available symbols/keys
- *grammar rules* : what is a valid expression.

e.g. + is a *binary operation*.

Syntax and variables

Focus of this PhD

Variables in expressions

$$(x + 5) \times y$$

x, y = **variables** = placeholders for other expressions
Substitution: variables $\mapsto$ expressions:

$$\begin{cases} \text{replace } x \text{ with } 3 \\ \text{replace } y \text{ with } z \times z \end{cases} \quad \leadsto \quad (3 + 5) \times (z \times z)$$

Bound variables and α-equivalence

α-**equivalence**:

$$x \mapsto 2 \times x \quad \text{should be identified with} \quad y \mapsto 2 \times y$$

"x is bound by $\mapsto$ in $x \mapsto 2 \times x$"

Syntax and recursion

Recursion (for syntax) = principle for investigating (step by step) a piece of valid syntactic data.

Examples of use of recursion

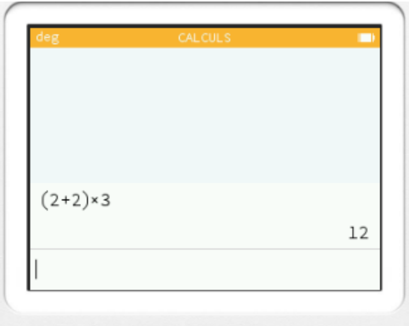
- count the number of operations in an arithmetic expression
- compute an arithmetic expression

What is a programming language?

Program execution

Program = valid *syntactic* text

Execution = modification of the program:



$$(2+2)\times 3 \xrightarrow{\text{1 execution step}} 4\times 3 \xrightarrow{\text{1 execution step}} 12$$

Operational semantics = description of how programs execute.

What is a programming language?

Finally

Programming language (PL) = syntax + operational semantics.

Specification of a PL = features uniquely characterizing a PL.

In 2 steps:

1. syntax

Example: specification of the syntax of arithmetic expressions

- numbers = constants
- $+$ and $\times$: operations expecting two expressions.

2. semantics

Caveat

There are **ineffective specifications**: no PL satisfies them.

Stupid example

Syntax with two constants 0 and 1 s.t. $0 = 1$ and $0 \neq 1$.

Related work

Non comprehensive list.

Specification of syntax

- Fiore's work: focus on **substitution**
  - syntactic operations (since [Fiore-Plotkin-Turi 1999])
  - syntactic equations [Fiore-Hur 2010] (e.g., $a + b = b + a$)
- Hirschowitz-Maggesi's work:
  - syntactic operations: a variant of Fiore's work
- *nominal* syntax: focus more on variable binding.

Specification of semantics

- SOS rules: λ-calculus with β-reduction still out of reach.
- Ahrens' work (2016): cannot deal with *non-congruent* reductions

This PhD builds upon Hirschowitz-Maggesi's and Ahrens' work.

Contributions of this PhD

1. mathematical definition of PL as **reduction monads**
2. specification of **syntactic equations**
   - (based on already known notion of specification of operations)
3. specification of **semantics**

using systematically the mathematical notion of *monads* and *modules* for taking care of substitution.

Articles

- CSL 2018 about 2.
- FSCD 2019 about 2. = variant of Fiore's approach.
- POPL 2020 about 1. and 3.

All in collaboration with Benedikt Ahrens, André Hirschowitz and Marco Maggesi.

Initial Semantics

Specification through initial semantics for justifying **recursion**.

$$\boxed{\text{Initiality} \Rightarrow \text{Recursion (and even induction)}}$$

Example: inductive sequences $S_{n+1} = f(S_n)$

We want $\mathbb{N} \xrightarrow{S} X$ s.t.

$$\begin{cases} S_0 = x_0 \\ S_{n+1} = f(S_n) \end{cases} (1) \qquad \text{where} \qquad \begin{array}{c} 1 \xrightarrow{x_0} X \\ X \xrightarrow{f} X \end{array} \quad \text{i.e.,} \quad 1 \amalg X \xrightarrow{[x_0, f]} X$$

$1 \coprod \mathbb{N} \xrightarrow{[0, \_+1]} \mathbb{N}$ is the **initial** algebra of $Y \mapsto 1 \amalg Y$. By initiality,

$$\exists! S : \mathbb{N} \to X$$

which is a morphism of algebras, i.e., satisfies (1).

Initial Semantics (in general)

Specification through initial semantics for justifying **recursion**.

1. Notion of **signature** for specifying stuff
   - **category of models** of a signature
   - initial model = specified object
   - initiality $\Rightarrow$ recursion
2. Ensure **effectiveness** of signatures = existence of the initial model.

On going related work

Initality project: proving initiality of a dependent type theory

This PhD: systematic treatment of substitution, in the untyped setting.

Outline

Outline

Ingredients

- Programming languages (PLs) as graphs
  - (**Syntax**) vertices = terms
  - (**Semantics**) arrows = reductions between terms
- Simultaneous substitution: variables $\mapsto$ terms
  - monads and modules over them

Example

λ-calculus with β-reduction:

- **Syntax:** $S, T \quad ::= x \mid S\ T \mid \lambda x.S$
- Modulo α-**equivalence**, e.g.

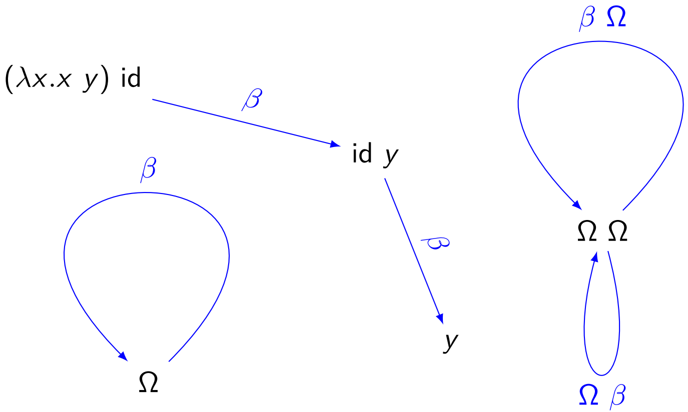
$$\lambda x.x = \lambda y.y$$

- **Reductions:** $(\lambda x.t)\, u \xrightarrow{\beta} t[x := u] \qquad + \quad$ congruences

Outline

PLs as graphs

Example: λ-calculus with β-reduction



- (**Syntax**) vertices = terms e.g. $\Omega = (\lambda x.x\,x)\,(\lambda x.x\,x)$
- (**Semantics**) arrows = reductions

Graphs

Definition

Graph $=$ a quadruple (A, V, σ, τ) where

$$A = \{\text{arrows}\} \qquad \sigma = \text{source of an arrow}$$

$$V = \{\text{vertices}\} \qquad \tau = \text{target of an arrow}$$

$$\begin{tikzcd} A \arrow[r, shift left, "\sigma"] \arrow[r, shift right, "\tau"'] & V \end{tikzcd}$$

$$\begin{array}{llll} \sigma : & A & \to & V \\ & t \xrightarrow{r} u & \mapsto & t \end{array} \qquad \begin{array}{llll} \tau : & A & \to & V \\ & t \xrightarrow{r} u & \mapsto & u \end{array}$$

$$\sigma(r) \xrightarrow{r} \tau(r)$$

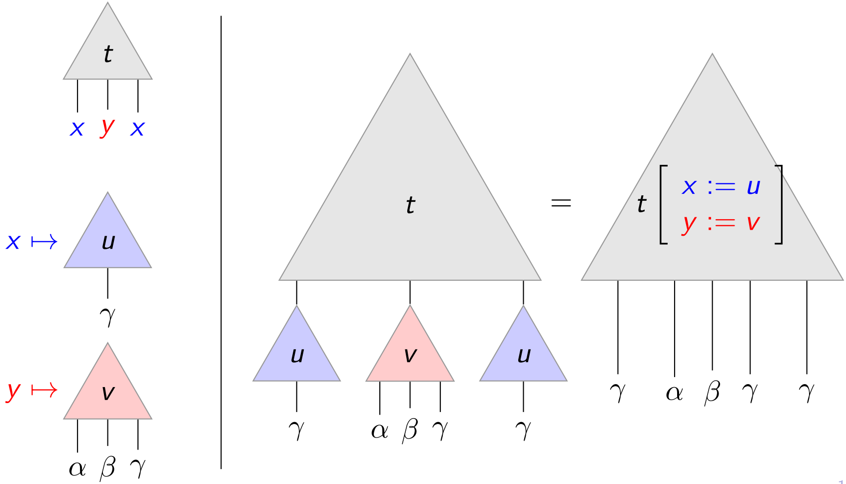
Outline

1. Reduction monads
   - Graphs
   - Substitution
2. Syntax
   - Operations
   - Equations
3. Semantics
   - Reduction rules
   - Reduction signatures

Simultaneous substitution

Syntax comes with substitution

terms (e.g. λ-terms) = trees with free variables as (distinguished) leaves.



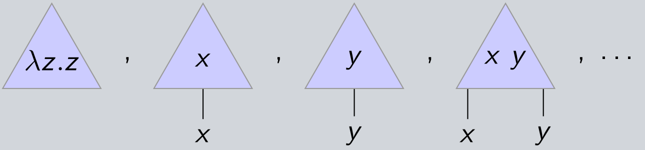
Simultaneous substitution made formal

Free variables indexing

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

Example: λ-calculus

$$L(\{x, y\}) = \Big\{ \quad , \quad , \quad , \quad , \cdots \Big\}$$



Simultaneous substitution

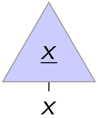
$$\forall f : X \to L(Y), \qquad \boxed{\begin{array}{rl} L(X) & \to L(Y) \\ t & \mapsto t[x \mapsto f(x)] \quad (\text{or } t[f]) \end{array}}$$

Monads capture simultaneous substitution

λ-calculus as a monad $(L, \_[\_], \eta)$

1. Simultaneous substitution $(L, \_[\_])$
2. Variables are terms

$$\begin{array}{cccc} \eta_X : & X & \to & L(X) \\ & x & \mapsto & \end{array}$$



3. Substitution laws:

$$\underline{x}[f] = f(x) \qquad\qquad t[x \mapsto \underline{x}] = t$$

$+$ associativity:

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

Substitution for semantics

We saw that syntax is expected to support substitution. This is also true of semantics.

Our notion of PL:

- **Syntax**: a monad $(L, \_[\_], \eta)$
- **Semantics**:
  - graphs $R(X) \underset{\tau_X}{\overset{\sigma_X}{\rightrightarrows}} L(X)$ for each X

$$R(X) = \begin{array}{c}\text{total set of reductions between}\\ \text{terms taking free variables in } X\end{array}$$

  - substitution of reduction: variables $\mapsto$ L-**terms**.

$$\frac{t \xrightarrow{r} u}{t[f] \xrightarrow{r[f]} u[f]}$$

Substitution for semantics made formal

R as a **module** over L

R supports L-monadic substitution:

$$\forall f : X \to \mathbf{L}(Y), \quad \boxed{\begin{array}{rl} R(X) & \to R(Y) \\ r & \mapsto r[x \mapsto f(x)] \quad (\text{or } r[f]) \end{array}}$$

$+$ substitution laws

Other examples of L-modules: L, $L \times L$, 1, ...

σ and τ as L-**module morphisms**

$$t \xrightarrow{r} u \leadsto t' \xrightarrow{r[f]} u' \quad \text{with} \quad \begin{cases} t' = t[f] \\ u' = u[f] \end{cases} \quad \text{i.e.,} \quad \begin{cases} \sigma(r[f]) = \sigma(r)[f] \\ \tau(r[f]) = \tau(r)[f] \end{cases}$$

Commutation with substitution $\Leftrightarrow$ Module morphisms $\sigma, \tau : R \to L$.

Reduction monads

Summary: graphs + substitution.

Definition

A **reduction monad** $R \overset{\sigma}{\underset{\tau}{\rightrightarrows}} T$ consists of

- T = monad (= module over itself)
- R = module over T
- $\sigma, \tau : R \to T$ are T-module morphisms.

Example

λ-calculus with β-reduction.

How can we specify a reduction monad?

1. signature for the (syntactic) operations for the monad;
2. reduction rules.

Outline

Overview

- Syntax = monad L
- Operations = module morphisms $\Sigma(L) \to L$
- 1-signatures specify operations
- 2-signatures specify operations + equations.

Outline

Operations as module morphisms

For any model of λ-calculus (in particular for L),

Application commutes with substitution

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

Categorical formulation

$L \times L$ supports L-substitution $\rightsquigarrow$ $L \times L$ is a **module over** L

application commutes with substitution $\rightsquigarrow$ $\mathrm{app} : L \times L \to L$ is a **module morphism**

[Hirschowitz-Maggesi 2007 : Modules over Monads and Linearity]

Examples of modules

We argued that syntactic operations are **module morphisms**. Basic examples of modules?

Module over a monad T: supports the T-monadic substitution

Examples

- T itself
- $M \times N$ for any modules M and N:

$$\forall (t, u) \in M(X) \times N(X), \qquad X \xrightarrow{f} T(Y),$$

$$\boxed{(t, u)[f] = (t[f], u[f])} \in M(Y) \times N(Y)$$

- M'= **derivative** of a module M:

$$M'(X) = M(\overbrace{X \amalg \{\diamond\}}^{X \text{ extended with a fresh variable } \diamond})$$

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

Operations as module morphisms

Operations can be combined into a single one.

Operations = module morphisms = maps commuting with substitution:

Example: λ-calculus

$$\begin{aligned} \mathrm{app} :&\quad L \times L \to L \\ \mathrm{abs} :&\quad L' \to L \end{aligned} \qquad \left\{ \begin{aligned} \mathrm{abs}_X : L(X \amalg \{\diamond\}) &\to L(X) \\ t &\mapsto \lambda \diamond . t \end{aligned} \right.$$

Combine operations into a single one:

$$\boxed{[\mathrm{app}, \mathrm{abs}] : (L \times L) \amalg L' \to L}$$

where (*coproducts* of modules M and N)

$$(M \amalg N)(X) = M(X) \amalg N(X)$$

1-signatures specify operations

Definition

A **1-signature** Σ is a (functorial) assignment

$$\boxed{T \mapsto \Sigma(T)}$$

monad module over T

Definition (model of a 1-signature Σ)

A **model** of Σ is a pair (T, m) denoted by $\Sigma(T) \xrightarrow{m} T$ s.t.

- T is a monad
- $\Sigma(T) \xrightarrow{m} T$ is a T-module morphism

Example: λ-calculus

$$[\mathrm{app}, \mathrm{abs}] : \Sigma_{LC}(L) \to L \qquad \text{where } \Sigma_{LC}(L) = (L \times L) \amalg L'$$

Syntax

We defined 1-signatures and their models. When is a signature effective?

(suitable notion of model morphism [Hirschowitz-Maggesi 2012])

Definition

The **syntax** specified by a 1-signature Σ is the initial object in its category of models.

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

Answer: No.

Counter-example: $\Sigma(R) = \mathcal{P} \circ R$

Powerset endofunctor on *Set*.

(for cardinality reasons)

Initial semantics for algebraic 1-signatures

We gave examples of effective 1-signatures. They were all **algebraic**.

Definition

Algebraic 1-signatures = 1-signatures built out of derivatives, finite products, disjoint unions, and the 1-signature $\Theta : T \mapsto T$.

Algebraic 1-signatures $\simeq$ binding signatures [Fiore-Plotkin-Turi 1999]
$\Rightarrow$ specification of n-ary operations, possibly binding variables.

Theorem (Fiore-Plotkin-Turi 1999)

Syntax exists for any algebraic 1-signature.

Example

λ-calculus

Question: Specify syntactic operations subject to some equations?

(*commutative associative* binary operation $+$ of diff. λ-calculus)

Quotient of algebraic signatures

We saw that algebraic signatures are effective. Can we specify effectively operations subject to equations?

Theorem (CSL 2018)

Syntax exists for any "quotient" of algebraic 1-signatures.

Example

a *commutative* binary operation $+$:

$$\forall a, b, \quad a + b = b + a$$


What about an **associative** operation?

Outline

Example: a commutative binary operation

Specification of a binary operation

1-signature	$T \mapsto T \times T$
model	$\begin{array}{c} T \times T \\ \downarrow + \\ T \end{array}$

Question What is an appropriate notion of model for a **commutative** binary operation?

- a monad T
- with a binary operation

$\Big\}$ a model $T \times T \xrightarrow{+} T$ of $\Theta \times \Theta$

- s.t.

$$\begin{tikzcd} T \times T \arrow[rr, bend left, "+"] \arrow[dr, bend right, "\mathrm{swap}"'] & & T \\ & T \times T \arrow[ur, bend right, "+"'] & \end{tikzcd}$$

where $\mathrm{swap}(t, u) = (u, t)$

Equations

Σ = 1-signature (e.g. binary operation $\Sigma(T) = T \times T$)

Definition

A **Σ-equation** $A \underset{v}{\overset{u}{\rightrightarrows}} B$ is a (functorial) assignment

$$\boxed{M = (\Sigma(T) \to T) \quad \mapsto \quad \left(A(M) \underset{v_M}{\overset{u_M}{\rightrightarrows}} B(M) \right)}$$

model of Σ

parallel pair of T-module morphisms

Example (Binary commutative operation)

$$\Sigma(T) = T \times T$$

$$\begin{tikzcd} T \times T \arrow[d, "+"] \\ T \end{tikzcd} \quad \mapsto \quad \begin{tikzcd} T \times T \arrow[rr, bend left, "+"] \arrow[dr, "swap"'] & & T \\ & T \times T \arrow[ur, "+"'] & \end{tikzcd}$$

2-signatures and their models

We defined equations. A set of equations yields a 2-signature.

Definition

A **2-signature** is a pair (Σ, E) where

- Σ is a 1-signature for monads
- E is a set of Σ-equations

Definition

A **model** of a 2-signature (Σ, E) consists of:

- a model $M = \begin{pmatrix} \Sigma(T) \\ \downarrow \\ T \end{pmatrix}$ of Σ s.t.

$$\forall\, A \overset{u}{\underset{v}{\rightrightarrows}} B \in E, \quad \boxed{u_M = v_M} : A(M) \to B(M)$$

morphism of models = morphisms as models of Σ.

Initial semantics for algebraic 2-signatures

We defined 2-signatures and their models. When is a 2-signature effective?

Theorem (FSCD 2019)

*Any **algebraic** 2-signature has an initial model.*

Definition

A 2-signature (Σ, E) is **algebraic** if:

- Σ is algebraic
- E consists of **elementary** Σ-equations

Main instances of elementary Σ-equations

$$A \rightrightarrows B \text{ s.t.} \qquad A\begin{pmatrix}\Sigma(T)\\ \downarrow \\ T\end{pmatrix} = \Phi(T) \qquad B\begin{pmatrix}\Sigma(T)\\ \downarrow \\ T\end{pmatrix} = T$$

for some *algebraic* 1-signature Φ.

(e.g. $\Phi(T) = T \times T$ for commutativity)

Example: algebraic 2-signature for differential λ-calculus

Lionel Vaux's version

$$\begin{array}{cccccccccc} L^d : & s, t & ::= & x \mid s\,t \mid \lambda x.s & \mid & Ds \cdot t & \mid & 0 & \mid & s + t \\ \Sigma_{LC^d}(T) & & = & \Sigma_{LC}(T) & \amalg & T \times T & \amalg & 1 & \amalg & T \times T \end{array}$$

Equations

- *associativity* and *commutativity* of $+$, neutrality of 0 for $+$
- bilinearity of $D\_ \cdot \_$ with respect to $+$, left linearity of application, linearity of abstraction

$$\lambda x.(s + t) = \lambda x.s + \lambda x.t \qquad \lambda x.0 = 0$$

Partial derivative $\frac{\partial \_}{\partial x} \cdot \_$ (usually defined by recursion on the syntax)

$$\boxed{D(\lambda x.s) \cdot t \to \lambda x. \left(\frac{\partial s}{\partial x} \cdot t \right)} \qquad \frac{\partial \_}{\partial x} \cdot \_ : T' \times T \to T'$$

Still specifiable as a 1-signature + recursive equations.

Outline

Specifying reduction monads

λ-calculus with (small-step) β-reduction as a reduction monad:

$$R \underset{\tau}{\overset{\sigma}{\rightrightarrows}} L$$

module over L

module morphisms

monad

- vertices $= L =$ initial model of the signature of λ-calculus.
- arrows $= R, \sigma, \tau = ?$
  - specified through *reduction rules* (to be made formal):

$$(\lambda x.t)\,u \to t[x := u] \qquad \frac{t \to t'}{t\;u \to t'\;u} \qquad \ldots$$

Outline

1. Reduction monads
   - Graphs
   - Substitution
2. Syntax
   - Operations
   - Equations
3. Semantics
   - Reduction rules
   - Reduction signatures

Analysis of a reduction rule

Example: binary congruence for application.

hypotheses

metavariables: $\underbrace{\text{as a } L\text{-module}}_{t,\, t',\, u,\, u'}$ L^4 $\mapsto$ $\boxed{\dfrac{t \to t' \qquad u \to u'}{t\ u \to t'\ u'}}$

conclusion

Hypothesis/conclusion = pair of λ-terms using metavariables

- as parallel module morphisms $L^4 \rightrightarrows L$

$$\text{e.g.,} \quad t\ u \to t'\ u' : \qquad \begin{array}{l} (t, t', u, u') \mapsto t\ u \\ (t, t', u, u') \mapsto t'\ u' \end{array}$$

- **Generalization**: $L \rightsquigarrow$ any model $\Sigma_{LC}(T) \to T$ of Σ_{LC}:

(application denoted by $\mathrm{app} : T \times T \to T$)

$$\text{e.g.,} \quad t\ u \to t'\ u' : \qquad \begin{array}{l} (t, t', u, u') \mapsto \mathrm{app}(t, u) \\ (t, t', u, u') \mapsto \mathrm{app}(t', u') \end{array}$$

Reduction rules

Definition

Let Σ = signature for monads (e.g. Σ_{LC} for congruence for application).

Definition of Σ-reduction rules

A Σ-**reduction rule** $(\vec{\sigma}, \vec{\tau})$

$$\boxed{\dfrac{\sigma_1 \to \tau_1 \quad \ldots \quad \sigma_n \to \tau_n}{\sigma_0 \to \tau_0}}$$

assigns (functorially) to each model $\Sigma(T) \to T$:

- $V(T)$ = T-module of metavariables (e.g. $V(T) = T^4$)
- parallel T-module morphisms $V(T) \underset{\tau_{i,T}}{\overset{\sigma_{i,T}}{\rightrightarrows}} T'^{\cdots\prime}$

We write

$\sigma_i, \tau_i : V \to \Theta^{(n_i)}$ $\qquad n_i$ = number of derivatives

Outline

Reduction signatures

Definition

A **reduction signature** is a pair $(\Sigma, \mathfrak{R})$ where

- Σ is a signature for monads (1- or 2-signature)
- $\mathfrak{R}$ is a family of Σ-reduction rules

Example: λ-calculus with β-reduction

- $\Sigma = \Sigma_{LC}$
- Σ-reduction rules:
  - β-reduction
  - congruence for application and **abstraction** ($T' \xrightarrow{\text{abs}} T$):

$$\frac{u \to u'}{\lambda x.u \to \lambda x.u'} \rightsquigarrow \frac{\pi_1 \to \pi_2}{\text{abs} \circ \pi_1 \to \text{abs} \circ \pi_2} \qquad \frac{T' \times T' \underset{\pi_{2,T}}{\overset{\pi_{1,T}}{\rightrightarrows}} T'}{T' \times T' \underset{\text{abs}\circ\pi_{2,T}}{\overset{\text{abs}\circ\pi_{1,T}}{\rightrightarrows}} T}$$

Models

We defined **reduction signatures**. What are their models?

A **model** of a signature $(\Sigma, \mathfrak{R})$ consists of:

- a reduction monad $R \overset{\sigma}{\underset{\tau}{\rightrightarrows}} T$ with a Σ-model structure on T
- for each reduction rule

$$\boxed{\dfrac{\sigma_1 \to \tau_1 \quad \dots \quad \sigma_n \to \tau_n}{\sigma_0 \to \tau_0}\, op} \qquad V \overset{\sigma_i}{\underset{\tau_i}{\rightrightarrows}} \Theta^{(n_i)} \quad \text{in } \mathfrak{R},$$

  - a mapping, for each $v \in V(T)(X)$,

$$\begin{pmatrix} \sigma_1(v) \xrightarrow{r_1} \tau_1(v) \\ \dots \\ \sigma_n(v) \xrightarrow{r_n} \tau_n(v) \end{pmatrix} \quad \mapsto \quad \sigma_0(v) \xrightarrow{op(r_1, \dots r_n)} \tau_0(v)$$

  - compatible with substitution:

$$op(r_1, \dots r_n)[f] = op(r_1[f], \dots, r_n[f])$$

Initiality

We defined **models** of a **reduction signature**. When is a signature effective?

(suitable notion of model morphism)

Theorem (POPL 2020)

Σ has an initial model (e.g. Σ is algebraic) $\Rightarrow$ $(\Sigma, \mathfrak{R})$ has an initial model.

Examples

- λ-calculus with small-step β-reduction
- λ-ex $=$ λ-calculus with explicit substitutions [Kesner 2009].

A Theory of Explicit Substitutions with Safe and Full Composition

Reduction signature for λ-ex

Syntax

λ-ex $=$ λ-calcul $+$ explicit substitution $t[x/u]$ s.t. x is bound in t:

$$\text{as a module morphism } L^{ex\prime} \times L^{ex} \to L^{ex}$$

subject to the equation

$$t[x/u][y/v] = t[y/v][x/u] \quad \text{if } y \notin fv(u) \text{ and } x \notin fv(v)$$

as a $\Sigma_{L^{ex}}$-equation $L^{ex\prime\prime} \times L^{ex} \times L^{ex} \rightrightarrows L^{ex}$.

Semantics

congruences, β-reduction $(\lambda x.t)\, u \to t[x/u]$, ...

$$t[x/u][y/v] \to t[y/v][x/u[y/v]] \quad \text{if } x \notin fv(u) \text{ and } y \in fv(u)$$

metavariable module: $L^{ex\prime\prime} \times L^{ex} \times L^{ex}_{\diamond}$

Extension of reduction monads

with associated effectivity theorem

1. Vertices: syntax/monad $\leadsto$ module of "configurations" over the syntax

Examples

- λ-calculus with small-step β-reduction cbv:
  - variables $\mapsto$ **values** (rather than terms)
  - Thus, monad of **values** (rather than terms)
  - Still, reductions between **terms** (rather than values) = "configurations" over the monad of values
- π-calculus
- differential λ-calculus (without its signature though)

2. Graph $\leadsto$ Bipartite graph

Example

λ-calculus with big-step β-reduction cbv: term $\to$ value.

Conclusion

Summary

- PLs as reduction monads
- Signatures for reduction monads with effectivity theorem

Perspectives

- Generalize the specification of vertices
  - specify the differential λ-calculus
- Generalize on the category of sets:
  - specify simply-typed PLs: category of families of sets (indexed by simple types)
  - specify Finster-Mimram's monad of weak ω-groupoids: category of globular sets
- Equations between reductions
  - relational reductions (at most 1 reduction between terms).

Thank you!