Mathematical specifications of programming languages

Ambroise Lafont[1]

[1]University of New South Wales
Sydney, Australia

October 20, 2020

That is the question

What is a programming language, mathematically?

- In the literature, no well-established consensus.

Differential λ-calculus [Ehrhard-Regnier 2003]

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

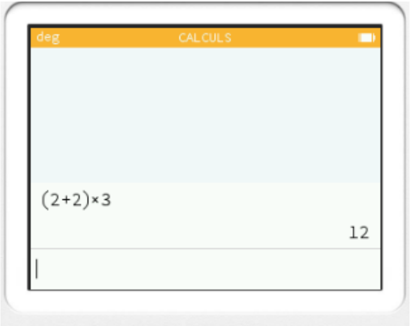
- In this talk:
  - a tentative notion of programming language, transition monads (FSCD 2020, with Tom and Andre Hirschowitz), and
  - a discipline for automatically generating well-behaved transition monads.
  - in the untyped case for ease of presentation (simply-typed case works as well)

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?

A graph whose vertices are programs.

$$
\begin{tikzcd}
(2+2)\times 3 \arrow[dd] & & & 3\times 2+2\times 2 \arrow[dl] \arrow[dr] & & x+(5+4) \arrow[dd] \\
 & & 6+2\times 2 \arrow[dr] & & 3\times 2+4 \arrow[dl] & \\
4\times 3 \arrow[dr] & & & 6+4 \arrow[d] & & x+9 \\
 & 12 & & 10 & &
\end{tikzcd}
$$

Variables = placeholders for expressions

- Substitution: $(x+(5+4))[x := 12] = 12+(5+4)$
- Reductions are stable under substitution

$$\frac{{\color{red}x}+(5+4) \longrightarrow {\color{red}x}+9}{{\color{red}12}+(5+4) \longrightarrow {\color{red}12}+9}.$$

$\rightsquigarrow$ Reduction monads!

A difficulty

Bound variables and α-equivalence

α-**equivalence**:

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

"x is bound by λ in $\lambda x.2 \times x$"

Specifying programming languages: **initial semantics**

- Constructing syntax and reductions may be complex (cf. differential λ-calculus).
- Often easier to describe the models.

Model $\approx$ graph with interpretation of the operations and reductions

a model of arithmetic expressions: $\mathbb{Z}$ (or rather $\mathbb{Z}[x, y, \ldots]$)

- Syntactic "+" $\rightsquigarrow$ actual "+" ,
- Syntactic "×" $\rightsquigarrow$ actual "×" , ...

- Programming language = initial model.
- Initiality $\Rightarrow$ recursion principle.

Notion of signature

- Specifies models.
- Effective iff the initial model exists.

Related work: syntax

Two main notions of syntax:

- Substitution monoids ($\approx$ finitary monads) [Fiore-Plotkin-Turi, 1999].
- Nominal sets [Gabbay-Pitts, 1999].

wider recursion principle more structured models

$[\mathbb{N}, \mathrm{Set}]$ nominal sets monads

This approach: monads

Related work: specifying syntax

Main notions of signature for monads:

- Pointed strong endofunctors [Fiore-Plotkin-Turi, 1999].
- Equational systems [Fiore-Hur, 2010].
- Modules [Hirschowitz-Maggesi, 2007].

This approach: modules

Related work: semantics

Semantic notions of programming language:

- Distributive laws [Plotkin-Turi, 1997].
- double categories [Meseguer, the Montanari school].

Do not cover higher-order languages.

- 2-categories [Power, Seely,...].
- relative monads [Ahrens, 2016].

Only covers congruent semantics.

In this talk

1. Mathematical definition of programming languages as **transition monads**, generalising **reduction monads** (POPL'20 with B. Ahrens, A. Hirschowitz, M. Maggesi).
2. Specification of **syntactic equations**, based on modules over monads.
3. Specification of **semantics**.

Systematic use of monads and modules for taking care of substitution.

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.

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

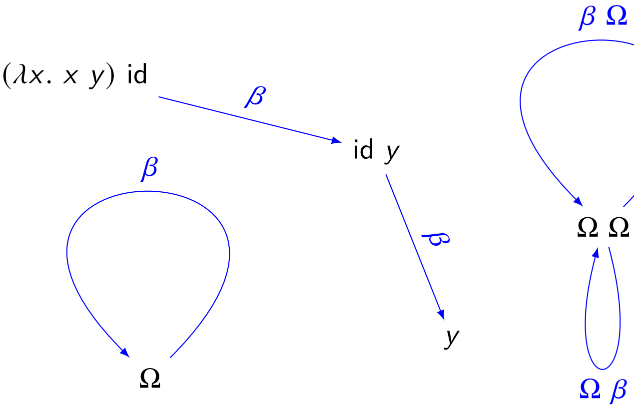
- **Reductions:** $(\lambda x.t)\ u \xrightarrow{\beta} t[x := u] \qquad + \quad$ congruences

Outline

1. Transition monads
   - Graphs
   - Substitution
2. Generating transition monads (Initial Semantics)
3. Examples

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

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

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

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

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

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

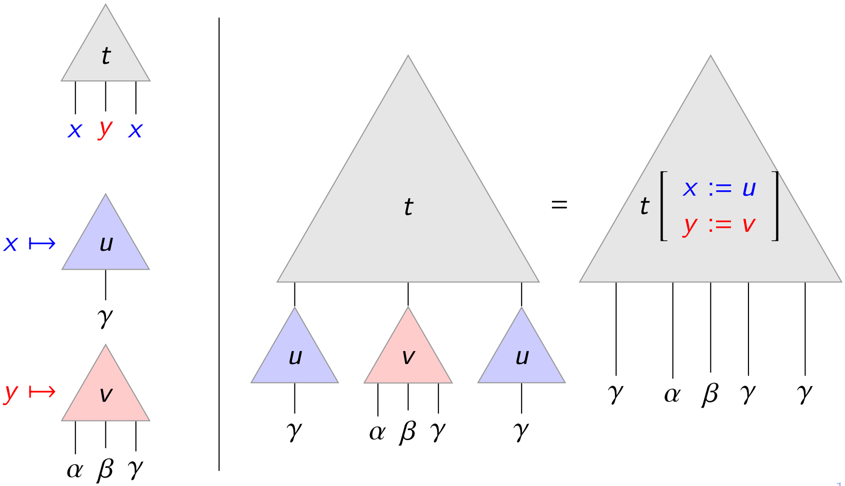
Outline

1. Transition monads
   - Graphs
   - Substitution
2. Generating transition monads (Initial Semantics)
3. Examples

Simultaneous substitution

Syntax comes with substitution

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



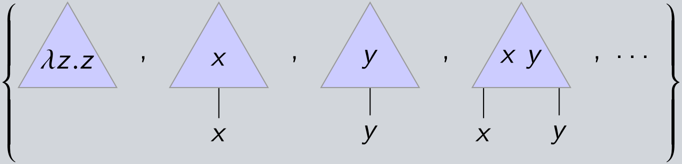
Simultaneous substitution made formal

Free variables indexing

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

Example: λ-calculus

$L(\{x, y\}) =$



Simultaneous substitution

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

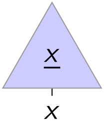
Monads model simultaneous substitution

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

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

$$\eta_X : \quad X \quad \to \quad L(X)$$

$$x \quad \mapsto$$



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

Syntax supports 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:

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

$+$ substitution laws

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

σ and τ as L-**module morphisms**

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

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

Reduction monads

Summary: graphs + substitution.

Definition

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

- Untyped case: base category = Set
- Simply-typed case: base category = Set^{Types}

Transition monads

Generalising cbv λ-calculus, and reduction monads

cbv λ-calculus (big-step)	*Values* (monad)	$\begin{array}{ccc} & \mathit{Transitions} & \\ \swarrow^{\mathit{source}} & & \searrow^{\mathit{target}} \\ \mathit{Terms} & & \mathit{Values} \end{array}$
transition monads	a monad T	T-module morphisms $M_1 \xleftarrow{\mathit{source}} \mathit{Trans} \xrightarrow{\mathit{target}} M_2$
reduction monads	a monad T	$T \xleftarrow{\mathit{source}} \mathit{Trans} \xrightarrow{\mathit{target}} T$

Examples: $\bar{\lambda}\mu$-calculus, cbv λ-calculus, π-calculus, differential λ-calculus, GSOS specification

Outline

1. Transition monads
   - Graphs
   - Substitution
2. Generating transition monads (Initial Semantics)
3. Examples

Constructing *transition monads*

We have a definition of programming languages as transition monads.

Can we construct them from *simple specifications*?

We provide:

- a notion of *simple specification* $=$ **signature** for transition monads
- a theorem ensuring the existence (unique up to iso) of a transition monad matching a spec

Three-level specification

Transition monad = $(T\ ,\ M_1 \xleftarrow{source} Trans \xrightarrow{target} M_2)$

Three spec steps:

Step	Component	Nature	Specification
1	T	monad	Operations + Equations
2	M_1, M_2	T-modules	Operations + Equations
3	$Trans$, $source$, $target$	"transition structure"	Transition rules as $\dfrac{t_1 \to u_1 \ldots t_n \to u_n}{t \to u}$

$\Rightarrow$ Three notions of signatures.

Outline

1. Transition monads
   - Graphs
   - Substitution
2. Generating transition monads (Initial Semantics)
3. Examples

Examples

$$\text{Transition monad} = (T\ ,\ M_1 \xleftarrow{source} Trans \xrightarrow{target} M_2)$$

Upcoming examples

1.	cbn λ-calculus	full signature (sketched)
2.	cbn λ-calculus	signature for T
3.	cbn λ-calculus	left congruence rule for application
4.	cbn λ-calculus	congruence rule for abstraction (involves a binding variable)
5.	cbv λ-calculus	signature for M_i
6.	differential λ-calculus	signature for M_i
7.	differential λ-calculus	signature for T

Example 1/7: small-step cbn λ-calculus

Transition monad $= (T, M_1 \xleftarrow{source} Trans \xrightarrow{target} M_2)$

Signature for cbn λ-calculus

Step	Component	Nature	Specification
1	T	monad	Operations = app, abs
2	M_1, M_2	T-modules	$M_1 = M_2 = T$
3	*Trans*, *source*, *target*	"transition structure"	β-rule + congruences

Example 2/7: Specify the monad of λ-terms

(untyped) cbn λ-calculus: $(T, T \xleftarrow{source} Trans \xrightarrow{target} T)$

- Syntax "generated" by

application	$T \times T \to T$	
λ-abstraction $\lambda x.t$	$T' \to T$	T' = module of terms depending on an extra variable
(variables)	$Var \to T$	

Signature for T

2 operations (application/abstraction)

- Monads always have variables: no need to specify them
- "operation"= *module morphism*: compatible with substitution:

$$(t_1\ t_2)[y \mapsto u_y] = t_1[y \mapsto u_y]\ t_2[y \mapsto u_y]$$

References "Second-order equational logic" Fiore-Hur '10,
"Modular specification of monads" Ahrens et al. '19

Disgression on T'

- 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.

$$\text{abs} : \quad L' \quad \to L \qquad \begin{cases} \text{abs}_X : L(X \amalg \{\diamond\}) \to L(X) \\ \hfill t \mapsto \lambda \diamond .t \end{cases}$$

Example 3/7: Left congruence for application

cbn λ-calculus: $(T, T \xleftarrow{source} Trans \xrightarrow{target} T)$

Left congruence rule for application

$$\frac{t_1 \longrightarrow t_2}{app(t_1, u) \longrightarrow app(t_2, u)}$$

- Easy interpretation of transition rules:

Components of the rule	Interpreted as...
3 "metavariables": t_1, t_2, u	a "metavariable" T-module $V = T \times T \times T$
1 "premise": $t_1 \longrightarrow t_2$	$V \longrightarrow M_1 \times M_2$ $(t_1, t_2, u) \mapsto (t_1, t_2)$ (T-module morphism)
"conclusion": $app(t_1, u) \longrightarrow app(t_2, u)$	$V \longrightarrow M_1 \times M_2$ $(t_1, t_2, u) \mapsto (\ app(t_1, u),\ app(t_2, u)\ )$

Example 4/7: Binding variables in rules

cbn λ-calculus: $(T, T \xleftarrow{source} Trans \xrightarrow{target} T)$

Congruence rule for abstraction

$$\frac{t_1 \to t_2}{\lambda x.t_1 \to \lambda x.t_2}$$

- "metavariables" t_1 and t_2: terms that may depend on x.
- $T' = T$-module of terms depending on an additional variable

Components of the rule	Interpreted as...
2 "metavariables": t_1,t_2	a "metavariable" T-module $V = T' \times T'$
1 "premise": $t_1 \to t_2$	$V \to T' \times T'$ $(t_1,t_2) \mapsto (t_1, t_2)$ (T-module morphism)
"conclusion": $\lambda x.t_1 \to \lambda x.t_2$	$V \to T \times T$ $(t_1, t_2) \mapsto (\lambda x.t_1, \lambda x.t_2)$

Example 5/7: Specify M_i for cbv

$$\text{Transition monad} = (T, \quad M_1 \xleftarrow{source} Trans \xrightarrow{target} M_2)$$

$$\text{cbv } \lambda\text{-calculus} = (Vals, Tms \xleftarrow{source} Trans \xrightarrow{target} Vals)$$

Syntax of values and terms

$$\begin{aligned} Vals : \ v, w &::= x | \lambda x.t \\ Tms : \ t, u &::= \underbrace{x | \lambda x.t}_{} | t\, u \\ &::= \quad v \quad | t\, u \end{aligned} \quad \Rightarrow \quad \begin{aligned} \text{terms} &= \text{binary trees } \textit{of} \text{ values} \\ Tms &= BinTree \circ Vals \end{aligned}$$

In fact, by definition of a transition monad,

- M_i is always of the shape $S_i \circ T$. Here,

  $T = Vals \qquad M_1 = BinTree \circ T \qquad M_2 = Id \circ T (= T)$

- Signature for M_i = Signature for S_i

Signature for *BinTree*

variables + 1 binary operation (accounts for $t\, u$ in Tms)

Example 6/7: Specify M_i for DLC

$$\text{Transition monad} = (T\ ,\ M_1 \xleftarrow{source} Trans \xrightarrow{target} M_2)$$

Differential λ-calculus (DLC)

Syntax monad T of terms (a variant of λ-calculus)

Semantics a $\underbrace{\text{term } t}_{M_1 = Id \circ T\ (=T)}$ reduces to a $\underbrace{\text{multiterm } t_1 + \cdots + t_n}$

multiterms = formal sum of terms

$M_2 = FormalSum \circ T$

Signature for *FormalSum*

Operations	a constant 0, a binary operation $+$, variables
Equations	commutativity, associativity, unitality

Example 7/7: the monad of DLC

differential λ-calculus: $(T, M_1 \xleftarrow{source} Trans \xrightarrow{target} M_2)$

- Syntax of DLC = variant of λ-calculus

Application of DLC

$$app : (t, U) \mapsto t\,U$$

$$\begin{aligned}
\text{input of } app &= \text{a term } t \text{ and a multi-term } U = u_1 + \cdots + u_n \\
&= \text{a term} \quad \text{and a formal sum } of \text{ terms} \\
\text{input } module \text{ of } app &= \quad T \quad \times \quad (FormalSum \circ T)
\end{aligned}$$

Signature for T

3 operations (no equation):

application $t\,U$	$T \times (FormalSum \circ T) \to T$
differential application $Dt \cdot u$	$T \times T \to T$
λ-abstraction	(as before)

Conclusion

Summary

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

Perspectives and other works

- Abstracting well-known theorems in this setting, e.g. Howe's method:
  - "A cellular Howe's theorem", LICS'20 with T. Hirschowitz and P. Borthelle.
  - can it be adapted to the setting of transition monads?
- Morphisms of transition monads = compilations
  - explore different variants of this definition, leading to different correctness criteria.
  - replay well-known examples in the setting of transition monads, e.g, Plotkin's CPS translations of λ-calculus.
- Effectful transitions?