Modules over monads and operational semantics

André Hirschowitz[1] Tom Hirschowitz[2] Ambroise Lafont[3]

[1]Université Côte d'Azur, CNRS, Nice, France

[2]Univ. Grenoble Alpes, Univ. Savoie Mont Blanc, CNRS, LAMA, France

[3]University of New South Wales, Australia

FSCD 2020 (prerecorded talk)

Summary

What is a programming language (syntax and semantics), mathematically?

This contribution:

- a notion of programming language: transition monads;
- a discipline for automatically generating them;
- generalises the reduction monads of Ahrens et al. (POPL 20)
  - new applications:

$\bar{\lambda}\mu$-calculus	π-calculus
cbv λ-calculus	differential λ-calculus
GSOS specifications	

  - simply-typed syntax (and semantics)

Related work

- *Reduction monads* of Ahrens et al. (POPL 2020)
  - × untyped
  - × cbv λ-calculus out of reach
  - – **this contribution** = a generalisation of *reduction monads*
- *Mathematical Operational Semantics* (Turi-Plotkin '97)
  - – Deeply developed
  - × Higher-order languages (such as λ-calculus) only starting to be investigated (Peressoti '17)
- Rewriting with variable binding (categorical approach)
  - – e.g. Hamana 2003, T. Hirschowitz 2013, Ahrens 2016
  - × only congruent transitions $\Rightarrow$ weak reduction out of reach

Outline

Basic components of a *transition monad*

Example: (small-step) cbv λ-calculus.
A **term** $t, u, \ldots$ reduces to a **value** $v, w, \ldots$

Monad of values

- *Variables* $\subset$ *Values*
- $v[\vec{w}/\vec{x}] \in$ *Values* (value substitution)

Modules of terms and transitions

- value substitution *in terms*:

$$t[\vec{w}/\vec{x}] \in Terms$$

- value substitution *in transitions*:

$$\frac{t \longrightarrow v}{t[\vec{w}/\vec{x}] \longrightarrow v[\vec{w}/\vec{x}]}$$

Generalising cbv λ-calculus, and reduction monads

cbv λ-calculus	*Values* (monad)	$Terms \xleftarrow{source} Transitions \xrightarrow{target} Values$
transition monads	a monad T	T-module morphisms $M_1 \xleftarrow{source} Trans \xrightarrow{target} M_2$
reduction monads (Ahrens et al.'20)	a monad T	$T \xleftarrow{source} Trans \xrightarrow{target} T$

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

Quick definition of T-modules
(given a monad T)

Components of a T-module M

- a functor M
- a natural transformation $MT \xrightarrow{\rho} M$
  $\Rightarrow$ **substitution** (see below)
- (+ some substitution laws)

Substitution for modules

- $m \in M(X) =$ "m takes free variables in X"
- $m[x \mapsto t_x] =$ "simultaneous substitution in m of every $x \in X$ with $t_x \in T(Y)$"

$$X \xrightarrow{t_\_} T(Y) \qquad \begin{array}{ccccc} M(X) & \xrightarrow{Mt_\_} & M(T(Y)) & \xrightarrow{\rho_Y} & M(Y) \\ m & \longmapsto & & & m[x \mapsto t_x] \end{array}$$

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 in the paper (one for each step), building upon

- *reduction monads*, Ahrens et al. POPL '20
- (steps 1-2) previous work by Fiore-Hur (*equational systems*) and Ahrens et al. (*signatures for monads*).

Examples

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

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

4 first examples already presented in Ahrens et al. '20 (cbn λ-calculus is a reduction monad)

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)

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[1] 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": $app(t_1, u) \longrightarrow app(t_2, u)$	$V \longrightarrow M_1 \times M_2$ $(t_1, t_2, u) \mapsto (app(t_1, u), app(t_2, u))$

[1] introduced by Ahrens et al. in the case of reduction monads.

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

$$Vals: \ v, w ::= x \mid \lambda x.t$$
$$Tms: \ t, u ::= \underbrace{x \mid \lambda x.t}_{} \mid t\,u$$
$$::= \quad v \quad \mid t\,u$$

$\Rightarrow$

terms = binary trees *of* values

$Tms = BinTree \circ Vals$

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

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

input of app = a term t and a multi-term $U = u_1 + \cdots + u_n$
= a term and a formal sum *of* terms
input *module* of app = $T \times (FormalSum \circ T)$

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)

Future work: strengthen the recursion principle

Initial semantics (general framework)

- specified object by a signature Σ = *initial object* in the category of *models* of Σ
- *initiality* $\Rightarrow$ recursion principle

transition monads reduction monads

- general models + general models

easier signatures stronger recursion principle

(Ahrens et al. '20)

Future work alternative notion of signatures with more general models (as in Ahrens et al. '20)

$\Rightarrow$ stronger recursion principle

A difficulty with general models à la Ahrens et al. '20: DLC

$(T, M_1 \leftarrow Trans \rightarrow M_2)$ specified by a 3-step signature

component	Σ_1	Σ_2	Σ_3 **(to be generalised)**
specifies	T	M_1, M_2	$\leftarrow Trans \rightarrow$

Future work

Σ_3 specifies transition rules for

T = ~~'the' initial~~ *any* model of Σ_1 (as in Ahrens et al. '20)

(M_1, M_2) = ~~'the' initial~~ *any* model of Σ_2

Specifying the transition rules of DLC

transitions involve intermediary syntactic constructions

T = 'the' **initial** model of Σ_1	T = any model of Σ_1
define them by **recursion**	recursion not available!

Conclusion

- A mild generalisation of Ahrens et al.'s *reduction monads* '20
  $\Longrightarrow$ new notion of programming language:

transition monads	a monad T	T-module morphisms $$\begin{array}{ccc} & \mathit{Trans} & \\ \swarrow^{\mathit{source}} & & \searrow^{\mathit{target}} \\ S_1 \circ T & & S_2 \circ T \end{array}$$

- Associated notion of specification
  - with easy interpretation of transition rules
- Numerous new examples:

$\bar{\lambda}\mu$-calculus	π-calculus	GSOS specifications
cbv λ-calculus	differential λ-calculus	