

Mathematical specification of programming languages using monads and modules over them

Ambroise Lafont¹

¹University of New South Wales
Sydney, Australia

October 26, 2020

That is the question

What is a programming language, mathematically?

- In the literature, no well-established consensus.

Differential λ -calculus [Ehrhard-Regnier 2003]

~10 pages (section 2 \rightarrow beginning of section 3) describing the programming language and proving some [properties](#).

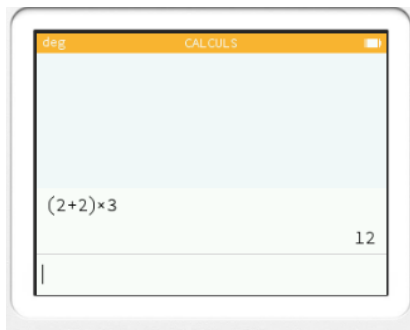
In this talk:

- 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?

2 components:

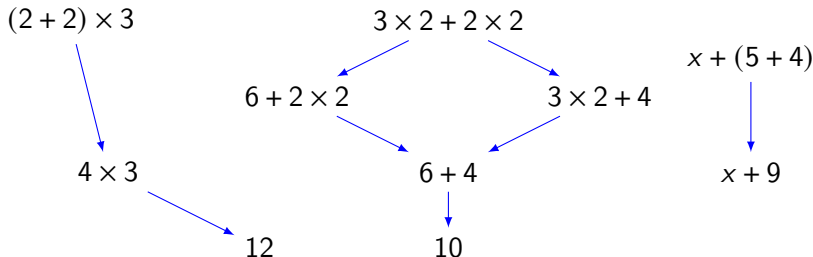
- **Syntax:** formal language for writing programs;
- **Operational semantics:** how do programs *execute*.



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

What is a programming language?

A graph whose vertices are programs.



Variables = placeholders for expressions

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

$$\frac{x + (5 + 4) \rightarrow x + 9}{12 + (5 + 4) \rightarrow 12 + 9}.$$

↪ Transition monads!

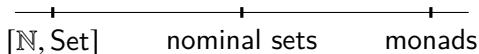
Related work: syntax

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



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

- Mathematical definition of programming languages as **transition monads**.
- Signatures for specifying them
- Systematic use of monads and modules for taking care of substitution.

Outline

- 1 Transition monads
 - Graphs
 - Substitution
 - Definition
- 2 Generating transition monads
 - Three-level specification
 - Examples
- 3 Generating compilations by initiality

Outline

- 1 Transition monads
 - Graphs
 - Substitution
 - Definition
- 2 Generating transition monads
 - Three-level specification
 - Examples
- 3 Generating compilations by initiality

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

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

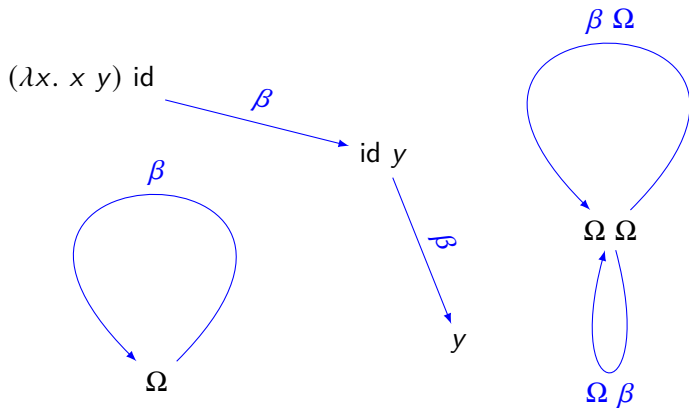
- **Reductions:** $(\lambda x. t) u \xrightarrow{\beta} t[x := u]$ + congruences

Outline

- 1 Transition monads
 - Graphs
 - Substitution
 - Definition
- 2 Generating transition monads
 - Three-level specification
 - Examples
- 3 Generating compilations by initiality

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

$A = \{\text{arrows}\}$

$\sigma = \text{source of an arrow}$

$V = \{\text{vertices}\}$

$\tau = \text{target of an arrow}$

$$A \begin{array}{c} \xrightarrow{\sigma} \\ \xrightarrow{\tau} \end{array} V$$

$$\sigma : \begin{array}{c} A \\ t \xrightarrow{r} u \end{array} \rightarrow V \quad \mapsto t$$

$$\tau : \begin{array}{c} A \\ t \xrightarrow{r} u \end{array} \rightarrow V \quad \mapsto u$$

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

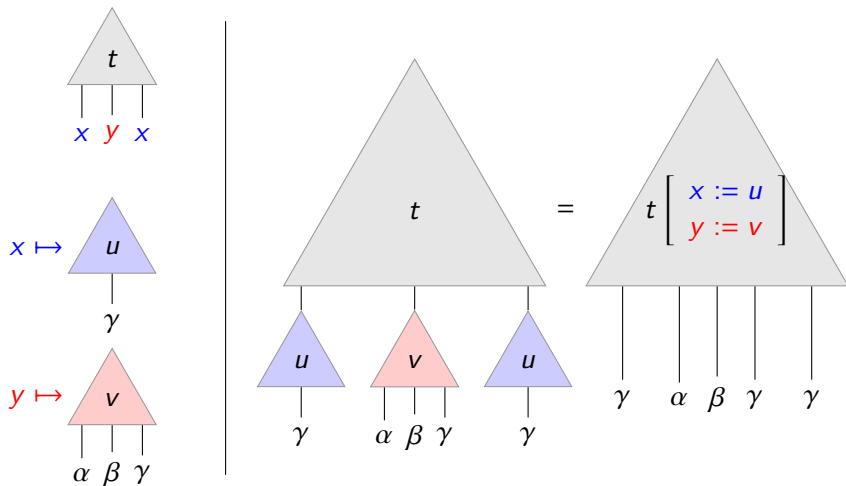
Outline

- 1 Transition monads
 - Graphs
 - Substitution
 - Definition
- 2 Generating transition monads
 - Three-level specification
 - Examples
- 3 Generating compilations by initiality

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

$$L(\{x, y\}) = \left\{ \begin{array}{c} \triangle \\ \lambda z. z \end{array} , \begin{array}{c} \triangle \\ x \\ | \\ x \end{array} , \begin{array}{c} \triangle \\ y \\ | \\ y \end{array} , \begin{array}{c} \triangle \\ x \ y \\ | \quad | \\ x \quad y \end{array} , \dots \right\}$$

Simultaneous substitution (bind)

$$\forall f : X \rightarrow L(Y),$$

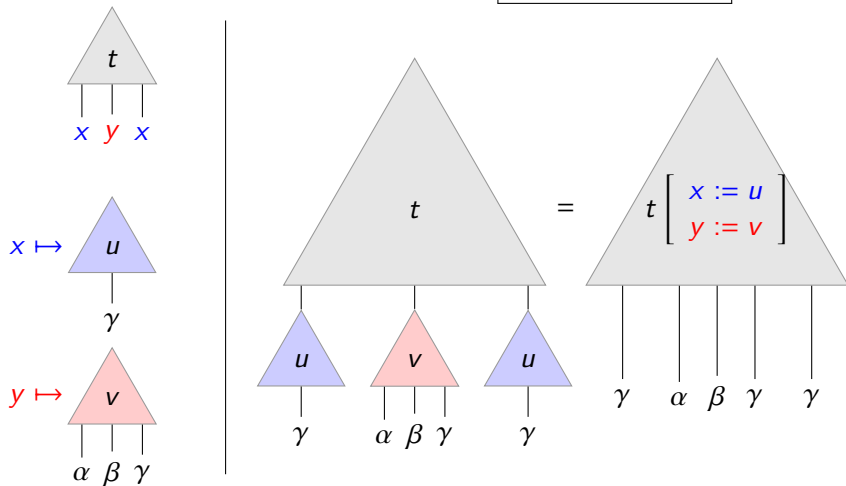
$$\begin{array}{l} L(X) \rightarrow L(Y) \\ t \mapsto t[x \mapsto f(x)] \quad (\text{or } t[f]) \end{array}$$

Simultaneous substitution

$$\forall f : X \rightarrow L(Y),$$

$$X = \{x, y\} \quad Y = \{\alpha, \beta, \gamma\}$$

$$\boxed{\begin{array}{l} L(X) \rightarrow L(Y) \\ t \mapsto t[f] \end{array}}$$

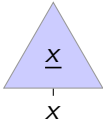


Monads model simultaneous substitution

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

① Simultaneous substitution $(L, _[_])$

② Variables are terms

$$\eta_X : X \rightarrow L(X)$$


$$x \mapsto \begin{array}{c} \triangle \\ \hline \underline{x} \\ | \\ x \end{array}$$

③ Substitution laws:

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

+ 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 $R(X) \xrightleftharpoons[\tau_X]{\sigma_X} L(X)$ for each X

$R(X) =$ total set of reductions between terms taking free variables in X

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

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

- $\Rightarrow R$ is a L -module, and σ, τ are module morphisms (see next slide)

Substitution for semantics made formal

R as a **module** over L

R supports L -monadic substitution:

$$\forall f : X \rightarrow L(Y),$$

$$\begin{array}{l} R(X) \rightarrow R(Y) \\ r \mapsto r[x \mapsto f(x)] \quad (\text{or } r[f]) \end{array}$$

+ substitution laws

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

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 $\sigma, \tau : R \rightarrow L$.

Outline

- 1 Transition monads
 - Graphs
 - Substitution
 - Definition
- 2 Generating transition monads
 - Three-level specification
 - Examples
- 3 Generating compilations by initiality

Transition monads (first attempt)

Summary: graphs + substitution.

Definition

A **transition monad** $R \xrightleftharpoons[\tau]{\sigma} T$ consists of

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

Example

λ -calculus with β -reduction.

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

What about big-step cbv λ -calculus? Terms reduce to values, not terms!

Transition monads

Generalising cbv λ -calculus, and reduction monads

cbv λ -calculus (big-step)	<i>Values</i> (monad)	
transition monads	a monad T	$M_1 \xleftarrow{\text{source}} \text{Trans} \xrightarrow{\text{target}} M_2$ <p>T-module morphisms</p>
reduction monads ¹	a monad T	$T \xleftarrow{\text{source}} \text{Trans} \xrightarrow{\text{target}} T$

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

¹POPL'20 with B.Ahrens, A. Hirschowitz, M. Maggesi.

Morphisms of transition monads

Simple case $M_i = T$

PLs	\Leftrightarrow	Transition monads
Compilations	\Leftrightarrow	Morphisms of transition monads

Morphism $(T \leftarrow Trans \rightarrow T) \longrightarrow (T' \leftarrow Trans' \rightarrow T') =$

(Syntax) A *monad morphism*¹ $T \xrightarrow{c} T'$

(Semantics) *Forward simulation*²: if $t_1 \xrightarrow{r} t_2$, then $c(t_1) \xrightarrow{\llbracket r \rrbracket} c(t_2)$

Examples (POPL'20, detailed later)

- λ -calculus + fixpoint op. $\longrightarrow \lambda$ -calculus
- λ -calculus + explicit substitution $t[x/u] \longrightarrow \lambda$ -calculus

¹mapping preserving substitution and variables

²backward simulations are often considered as a correctness criteria

Outline

- 1 Transition monads
 - Graphs
 - Substitution
 - Definition
- 2 Generating transition monads
 - Three-level specification
 - Examples
- 3 Generating compilations by initiality

Outline

- 1 Transition monads
 - Graphs
 - Substitution
 - Definition
- 2 Generating transition monads
 - Three-level specification
 - Examples
- 3 Generating compilations by initiality

Constructing *transition monads*

programming language = **transition monad**.

Can we construct them from *simple specifications*?

Overview

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

Specification through **initial semantics**

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

Model \approx transition monad + interpretation of the operations and reductions

Initial Semantics

To each signature is associated

- a notion of model = transition monad + additional structure
- a notion of morphism of models = compilation preserving additional structure
- a proof that the category of models has an initial object = object specified by the signature

Initiality \Rightarrow **recursion principle**.

Three-level specification

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

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 $\frac{t_1 \rightarrow u_1 \dots t_n \rightarrow u_n}{t \rightarrow u}$

\Rightarrow Three notions of signatures.

Outline

- 1 Transition monads
 - Graphs
 - Substitution
 - Definition
- 2 Generating transition monads
 - Three-level specification
 - Examples
- 3 Generating compilations by initiality

Examples

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

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

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	$\text{Trans},$ $\text{source},$ target	“transition structure”	β -rule + congruences

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

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

- Syntax “generated” by

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

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'

- $M' = \mathbf{derivative}$ of a module M :

X extended with a fresh variable x

$$M'(X) = M(\overbrace{X \amalg \{x\}})$$

used to model an operation binding a variable.

$$\text{abs} : L' \rightarrow L \quad \left\{ \begin{array}{l} \text{abs}_X : L(X \amalg \{x\}) \rightarrow L(X) \\ t \mapsto \lambda x. t \end{array} \right.$$

Example 3/7: Left congruence for application

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

Left congruence rule for application

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

- 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”: $t_1 \rightarrow t_2$	$V \rightarrow M_1 \times M_2$ (T -module morphism) $(t_1, t_2, u) \mapsto (t_1, t_2)$
“conclusion”: $\text{app}(t_1, u) \rightarrow \text{app}(t_2, u)$	$V \rightarrow M_1 \times M_2$ $(t_1, t_2, u) \mapsto (\text{app}(t_1, u), \text{app}(t_2, u))$

Example 4/7: Binding variables in rules

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

Congruence rule for abstraction

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

- “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”: $t_1 \rightarrow t_2$	$V \rightarrow T' \times T'$ (T -module morphism) $(t_1, t_2) \mapsto (t_1, t_2)$
“conclusion”: $\lambda x. t_1 \rightarrow \lambda x. t_2$	$V \rightarrow T \times T$ $(t_1, t_2) \mapsto (\lambda x. t_1, \lambda x. t_2)$

Example 5/7: Specify M_i for cbv

$$\begin{aligned}\text{Transition monad} &= (T, \quad M_1 \xleftarrow{\text{source}} \text{Trans} \xrightarrow{\text{target}} M_2) \\ \text{cbv } \lambda\text{-calculus} &= (\text{Vals}, Tms \xleftarrow{\text{source}} \text{Trans} \xrightarrow{\text{target}} \text{Vals})\end{aligned}$$

Syntax of values and terms

$\text{Vals} : v, w ::= x \mid \lambda x. t$

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

In fact, by definition of a transition monad,

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

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

- Signature for M_i = Signature for S_i

Signature for *BinTree*

variables (= labelled leaves) + 1 binary operation (building nodes)

Example 6/7: Specify M_i for DLC

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

Differential λ -calculus (DLC)

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

Semantics a term t reduces to a multiterm $t_1 + \dots + t_n$

$M_1 = \text{Id} \circ T (=T)$

multiterms = **formal sum** of terms

$M_2 = \text{FormalSum} \circ T$

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

- 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 $U = u_1 + \dots + u_n$
 = a term and a formal sum of terms

$$\text{input module of } app = T \times (FormalSum \circ T)$$

Signature for T

3 operations (no equation):

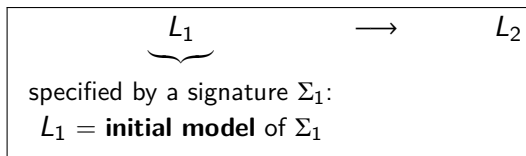
application $t \ U$	$T \times (\text{FormalSum} \circ T) \rightarrow T$
differential application $Dt \cdot u$	$T \times T \rightarrow T$
λ -abstraction	(as before)

Outline

- 1 Transition monads
 - Graphs
 - Substitution
 - Definition
- 2 Generating transition monads
 - Three-level specification
 - Examples
- 3 Generating compilations by initiality

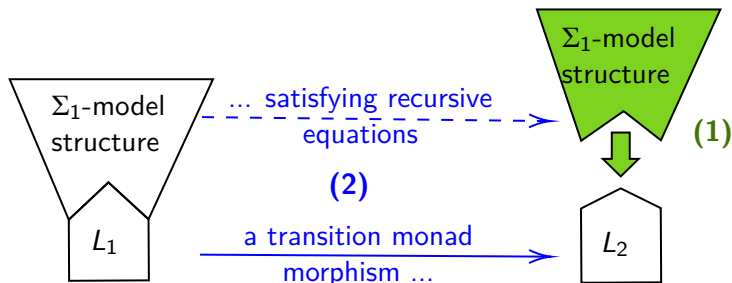
Generating compilations by initiality

Initiality \approx recursion principle



Data generating a compilation: a Σ_1 -model structure for L_2

\Rightarrow By recursion/**initiality**, get a model morphism $L_1 \rightarrow L_2$



Examples

$$\underbrace{L_1} \longrightarrow L_2$$

specified by a signature Σ_1 :

Recipe:

- ① provide a Σ_1 -model structure for L_2
- ② as a model morphism, the induced compilation satisfies recursive equations.

Upcoming examples (POPL'20)

- λ -calculus + formal fixpoint op. $\longrightarrow \lambda$ -calculus
 - ① construct a fixpoint operator in λ -calculus
 - ② formal fixpoint operator \mapsto constructed fixpoint operator
- λ -calculus + explicit substitution¹ $t[x/u] \longrightarrow \lambda$ -calculus
 - ① consider λ -calculus with its unary substitution operation
 - ② explicit substitution \mapsto real substitution

¹A Theory of Explicit Substitutions with Safe and Full Composition, [Kesner 2009]

Example 1/2: compiling λ -calculus + formal fixpoint op.

$$\underbrace{L_{\text{fix}}}_{\text{specified by } \Sigma_L + \Sigma_{\text{fix}}} \longrightarrow \underbrace{L}_{\text{specified by } \Sigma_L} \quad (\lambda\text{-calculus})$$

Signature Σ_{fix} specifying a fixpoint operator

- an operation $T' \xrightarrow{\text{fix}} T$
- reductions $\text{fix}(t) \rightarrow t[\underbrace{x}_{\text{the fresh variable}} \mapsto \text{fix}(t)]$

Model structure on L for Σ_{fix} (\Rightarrow compilation $L_{\text{fix}} \rightarrow L$)

- choose a fixpoint combinator: a term Y s.t. $Y u \rightarrow_{\beta}^* u (Y u)$
- define $\text{fix}(t) := Y(\lambda x.t)$

$$\underbrace{Y(\lambda x.t)}_{\text{fix}(t)} \rightarrow_{\beta}^* (\lambda x.t)(Y(\lambda x.t)) \rightarrow_{\beta} \underbrace{t[x \mapsto Y(\lambda x.t)]}_{t[x \mapsto \text{fix}(t)]}$$

Example 2/2: compiling λ -calculus + explicit substitution

$$\underbrace{L_{ex}}_{\text{specified by } \Sigma_L \setminus \{\beta\} + \Sigma_{ex}} \longrightarrow \underbrace{L}_{\text{specified by } \Sigma_L} \quad (\lambda\text{-calculus})$$

Signature Σ_{ex} for the explicit substitution

- an operation $T' \times T \xrightarrow{(t,u) \mapsto t[x/u]} T$ s.t.

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

- β -reduction $\boxed{(\lambda x. t)u \rightarrow t[x/u]}$ + congruences +

$$t[x/u][y/v] \rightarrow t[y/v][x/u[y/v]] \quad x \notin \text{fv}(v), y \in \text{fv}(u) \quad (1)$$

Model structure on L for Σ_{ex} (\Rightarrow compilation $L_{ex} \rightarrow L$)

- use the real substitution $T' \times T \xrightarrow{(t,u) \mapsto t[x:=u]} T$
- β -reduction + congruences + reflexive reduction (1)

Perspectives

- Generalise well-known theorems, e.g. Howe's method:
 - “A cellular Howe's theorem”, LICS'20 with T. Hirschowitz and P. Borthelle, in a simpler setting.
- Morphisms of transition monads = compilations
 - explore different variants (different correctness criteria).
 - try “academic” examples, e.g. Plotkin's CPS translations of λ -calculus.
 - “effective” Coq formalization (theory already formalized using UniMath for the syntax)
- Effectful transitions?