Mathematical specification of programming languages using monads and modules over them

Ambroise Lafont[1]

[1]University of New South Wales
Sydney, Australia

October 27, 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)

Ambroise Lafont   Specifying programming languages

What is a programming language?
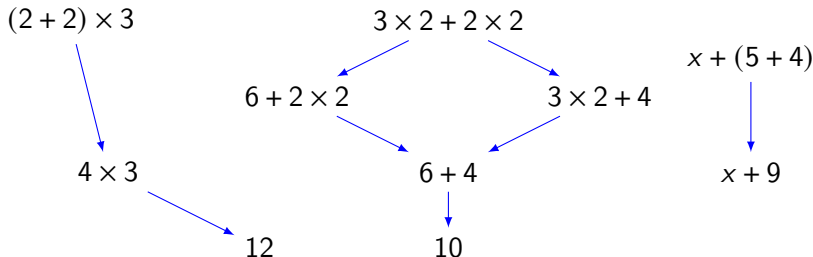
2 components:

- **Syntax**: formal language for writing programs;
- **Operational semantics**: how do programs *execute*.
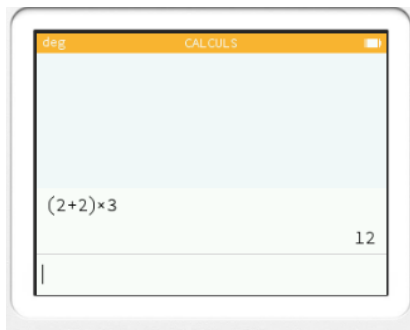
$$(2 + 2) \times 3 \xrightarrow{\text{1 execution step}} 4 \times 3 \xrightarrow{\text{1 execution step}} 12$$

Ambroise Lafont   Specifying programming languages

What is a programming language?
A graph whose vertices are programs.

$(2+2) \times 3$

$3 \times 2 + 2 \times 2$

$x + (5+4)$

$6 + 2 \times 2$

$3 \times 2 + 4$

4×3

$6 + 4$

$x + 9$

12

10

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

- *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

Ambroise Lafont   Specifying programming languages

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.

Ambroise Lafont   Specifying programming languages

# Outline

1. Transition monads
   - Substitution
   - Graphs
   - Definition

2. Generating transition monads
   - Three-level specification
   - Examples

3. Generating compilations by initiality

4. Conclusion

Outline

Ambroise Lafont   Specifying programming languages

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:** $\quad\quad S, T \quad ::= x \mid S\,T \mid \lambda x.S$

- Modulo α-**equivalence**, e.g.

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

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

Ambroise Lafont   Specifying programming languages

Outline

Ambroise Lafont   Specifying programming languages

Simultaneous substitution
Syntax comes with substitution

terms as syntactic derivation trees (free variables as distinguished leaves).
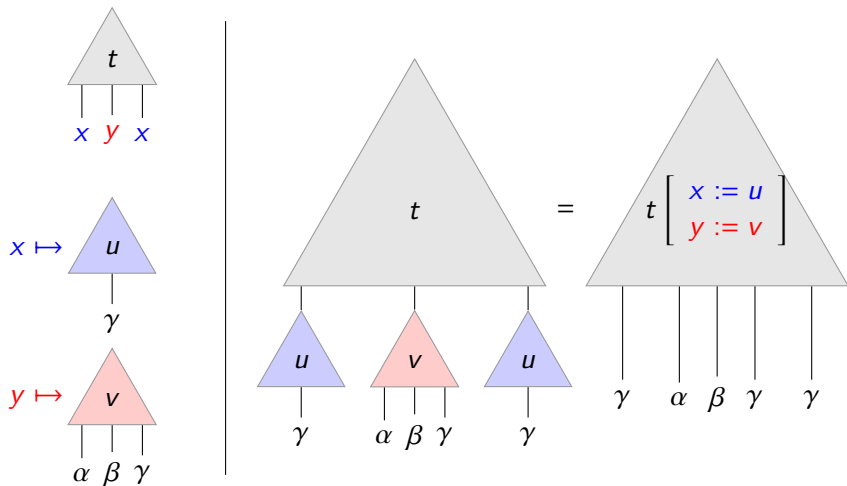
Simultaneous substitution made formal

Free variables indexing

$$L(X) = \{\text{terms taking free variables in } X\}$$

Example: λ-calculus

$$L(\{x, y\}) = \left\{ \begin{array}{c} \lambda z.z \end{array} \quad , \quad \begin{array}{c} x \\ \mid \\ x \end{array} \quad , \quad \begin{array}{c} y \\ \mid \\ y \end{array} \quad , \quad \begin{array}{c} x \; y \\ \mid \quad \mid \\ x \quad y \end{array} \quad , \quad \cdots \right\}$$

Simultaneous substitution (bind)

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

Ambroise Lafont   Specifying programming languages

Simultaneous substitution

$$\forall f : X \rightarrow L(Y),$$
$$X = \{x, y\} \qquad Y = \{\alpha, \beta, \gamma\}$$

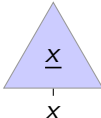$$\begin{array}{rl} L(X) & \rightarrow L(Y) \\ t & \mapsto t[f] \end{array}$$
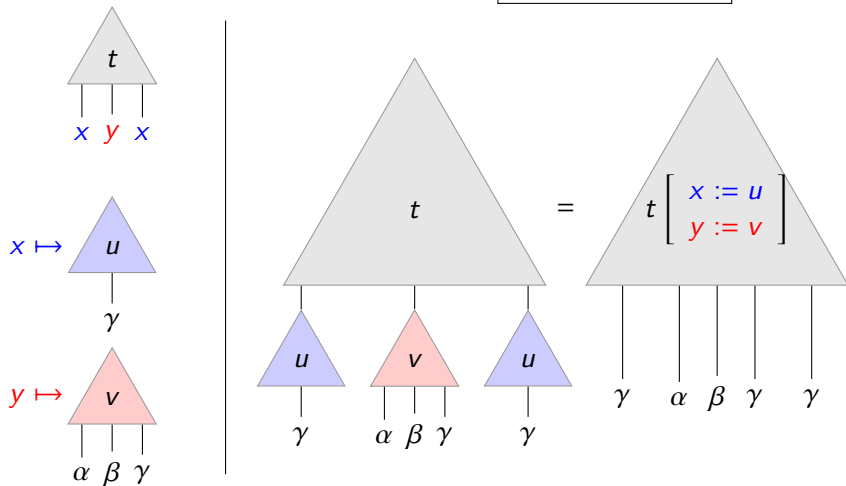
Monads model simultaneous substitution
λ-calculus as a monad $(L, \_[\_], \eta)$

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

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

$$x \quad \mapsto \quad \underline{x}$$

3. Substitution laws:

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

+ associativity:

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

Ambroise Lafont   Specifying programming languages

Outline

Ambroise Lafont   Specifying programming languages

PLs as graphs

Example: λ-calculus with β-reduction
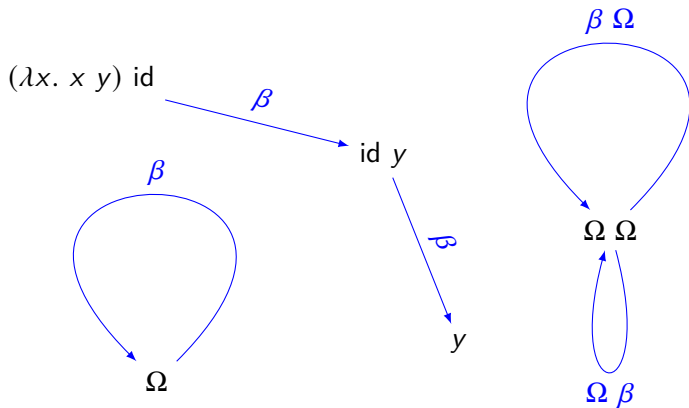
- (**Syntax**) vertices = terms e.g. $\Omega = (\lambda x.x\,x)\,(\lambda x.x\,x)$
- (**Semantics**) arrows = reductions

Graphs
Definition

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

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

$$A \underset{target}{\overset{source}{\rightrightarrows}} V$$

$$\sigma : \quad A \quad \to V \qquad \tau : \quad A \quad \to V$$
$$t \overset{r}{\to} u \;\mapsto t \qquad\qquad t \overset{r}{\to} u \;\mapsto u$$

$$\sigma(r) \overset{r}{\to} \tau(r)$$

Substitution for semantics
Syntax supports substitution. This is also true of semantics.

Our notion of PL:

- **Syntax**: a monad $(L, \_[\_], \eta)$
- **Semantics**:

    - graphs $R(X) \underset{target_X}{\overset{source_X}{\rightrightarrows}} L(X)$ for each X

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

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

    $$\frac{t \overset{r}{\to} u}{t[f] \xrightarrow{r[f]} u[f]} \qquad f : X \to L(Y)$$

$\Rightarrow$ R is a L-module, and *source*, *target* are module morphisms (see next slide)

Ambroise Lafont   Specifying programming languages

Substitution for semantics made formal

R as a **module** over L

R supports L-monadic substitution:

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

$+$ substitution laws

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

σ and τ as L-**module morphisms**

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

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

Outline

Transition monads (first attempt)

Summary: graphs + substitution.

Definition

A **transition monad** $R \xrightarrow[\tau]{\sigma} T$ 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}

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

Ambroise Lafont   Specifying programming languages

Transition monads
Generalising cbv λ-calculus, and reduction monads

cbv λ-calculus (big-step)	Values (monad)	Transitions	
transition monads	a monad T	T-module morphisms $M_1 \xleftarrow{source} Trans \xrightarrow{target} M_2$ (bipartite graph)	
reduction monads[1]	a monad T	$T \xleftarrow{source} Trans \xrightarrow{target} T$	

In the first row cell (Transitions):

$$\text{Terms} \xleftarrow{source} \text{Transitions} \xrightarrow{target} \text{Values}$$

Examples:
$\overline{\lambda}\mu$-calculus	π-calculus	differential λ-calculus
cbv λ-calculus	GSOS specs	computational λ-calculus[2]

---

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

[2]"Effectful Applicative Bisimilarity: Monads, Relators, and Howe's Method", Lago-Gavazzo-Levy LICS 2017

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

---

[1] mapping preserving substitution and variables

[2] backward simulations are often considered as a correctness criteria

Ambroise Lafont   Specifying programming languages

Outline

Outline

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

Ambroise Lafont   Specifying programming languages

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 additionnal 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 , $M_1 \xleftarrow{\textit{source}} \textit{Trans} \xrightarrow{\textit{target}} M_2$)

Three spec steps:

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

$\Rightarrow$ Three notions of signatures.

Outline

1. [Transition monads](#)
   - [Substitution](#)
   - [Graphs](#)
   - [Definition](#)

2. [Generating transition monads](#)
   - [Three-level specification](#)
   - [Examples](#)

3. [Generating compilations by initiality](#)

4. [Conclusion](#)

Examples

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

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

Ambroise Lafont   Specifying programming languages

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

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

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	$Trans$, $source$, $target$	"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:

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

used to model an operation binding a variable.

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

Ambroise Lafont   Specifying programming languages

Example 3/7: Left congruence for application

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

Left congruence rule for application

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

- 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": $t_1 \to t_2$	$V \to M_1 \times M_2$ $(t_1, t_2, u) \mapsto (t_1, t_2)$	(T-module morphism)
"conclusion": $app(t_1, u) \to app(t_2, u)$	$V \to M_1 \times M_2$ $(t_1, t_2, u) \mapsto (app(t_1, u), app(t_2, u))$	

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-module morphism) $(t_1, t_2) \mapsto (t_1, t_2)$
"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

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

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

Syntax of values and terms

$Vals : v, w ::= x | \lambda x.t$

$Tms : t, u ::= \underbrace{x | \lambda x.t}_{v} | t\,u \quad \Rightarrow \quad$ terms $=$ binary trees *of* values

$::= \quad v \quad | t\,u \qquad\qquad\qquad Tms = BinTree \circ Vals$

In fact, by definition of a transition monad,

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

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

- Signature for M_i = Signature for S_i

Signature for *BinTree*

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

Ambroise Lafont — Specifying programming languages

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

                                    multiterms = formal sum of terms

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

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

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

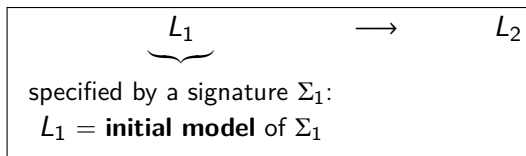### Signature for T

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)

Outline

Generating compilations by initiality
Initiality ≈ recursion principle

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

specified by a signature Σ_1:
   $L_1 = $ **initial model** of Σ_1

Data generating a compilation: a Σ_1-model structure for L_2
⟹ By recursion/**initiality**, get a model morphism $L_1 \to L_2$
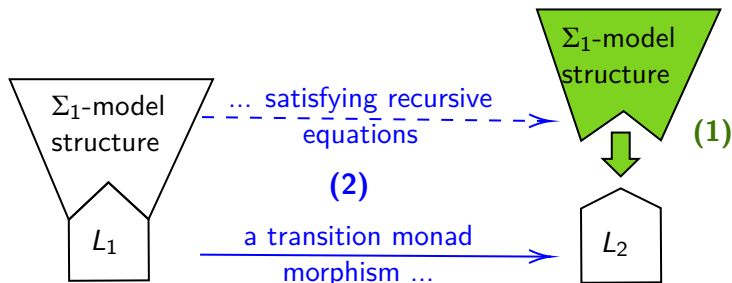
Examples

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

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

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

Ambroise Lafont   Specifying programming languages

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

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

Signature Σ_{fix} specifying a fixpoint operator

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

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

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

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

Ambroise Lafont   Specifying programming languages

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

$$\underbrace{L_{ex}} \qquad \longrightarrow \qquad \underbrace{L} \qquad (\lambda\text{-calculus})$$

specified by $"\Sigma_L \backslash \{\beta\} + \Sigma_{ex}"$ specified by Σ_L

Signature Σ_{ex} for the explicit substitution

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

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

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

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

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

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

Outline

Ambroise Lafont   Specifying programming languages

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 different setting.
  - "Effectful Applicative Bisimilarity: Monads, Relators, and Howe's Method", Lago-Gavazzo-Levy LICS 2017, for particular cases of transition monads (computational λ-calculus).
- 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)
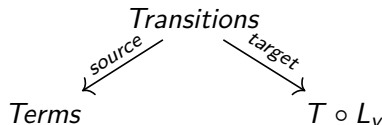
Example: computational λ-calculus[1]

Parameterized by:

- a set Σ of operations σ with specified arities
- a monad T with operations $T \times \cdots \times T \xrightarrow{\sigma^T} T$.

$$M, N \quad ::= \quad \text{return } V \mid VW \mid M \text{ to } x.N \mid \sigma(M, \ldots, M);$$
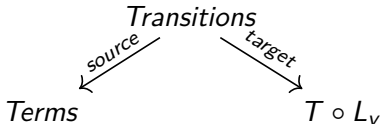$$V, W \quad ::= \quad x \mid \lambda x.M.$$

$\Rightarrow$ a monad L_v of **values** + a L_v-module of terms

$$
\begin{array}{ccc}
 & Transitions & \\
 & \swarrow \text{\tiny source} \qquad \text{\tiny target} \searrow & \\
Terms & & T \circ L_v
\end{array}
$$

---

[1]*Effectful Applicative Bisimilarity: Monads, Relators, and Howe's Method*, Lago-Gavazzo-Levy LICS 2017

Example: computational λ-calculus[1]
Semantics

$$
\begin{array}{ccc}
 & \textit{Transitions} & \\
 & \overset{\textit{source}}{\swarrow} \quad \overset{\textit{target}}{\searrow} & \\
\textit{Terms} & & T \circ L_v
\end{array}
$$

$$
\frac{}{\text{return } V \Downarrow \eta(V)} \text{ (ret)}
$$

$$
\frac{M \Downarrow X \qquad N[x := V] \Downarrow Y_V}{M \text{ to } x.N \Downarrow X \text{\ding{{>}}=}(V \mapsto Y_V)} \text{ (seq)}
$$

$$
\frac{M[x := V] \Downarrow X}{(\lambda x.M)V \Downarrow X} \text{ (app)}
$$

$$
\frac{M_1 \Downarrow X_1 \qquad \ldots \qquad M_k \Downarrow X_k}{\sigma(M_1, \ldots, M_k) \Downarrow \sigma^T(X_1, \ldots, X_k)} \text{ (op)}
$$

[1] *Effectful Applicative Bisimilarity: Monads, Relators, and Howe's Method*,
Lago-Gavazzo-Levy LICS 2017