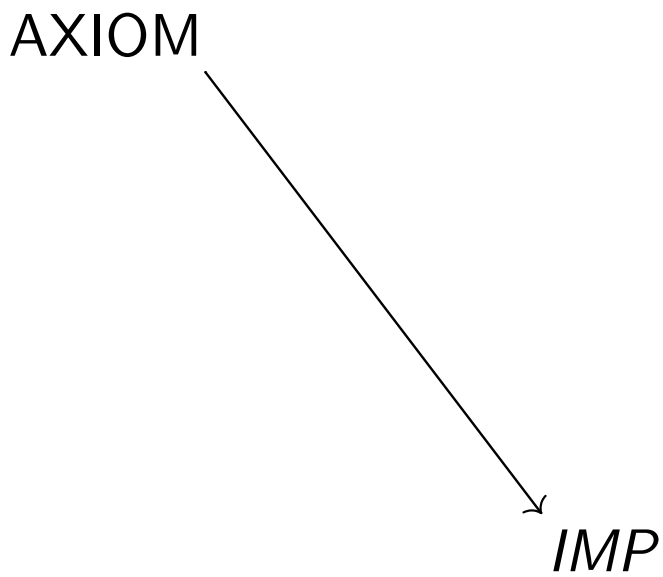
From AXIOM down to IMP

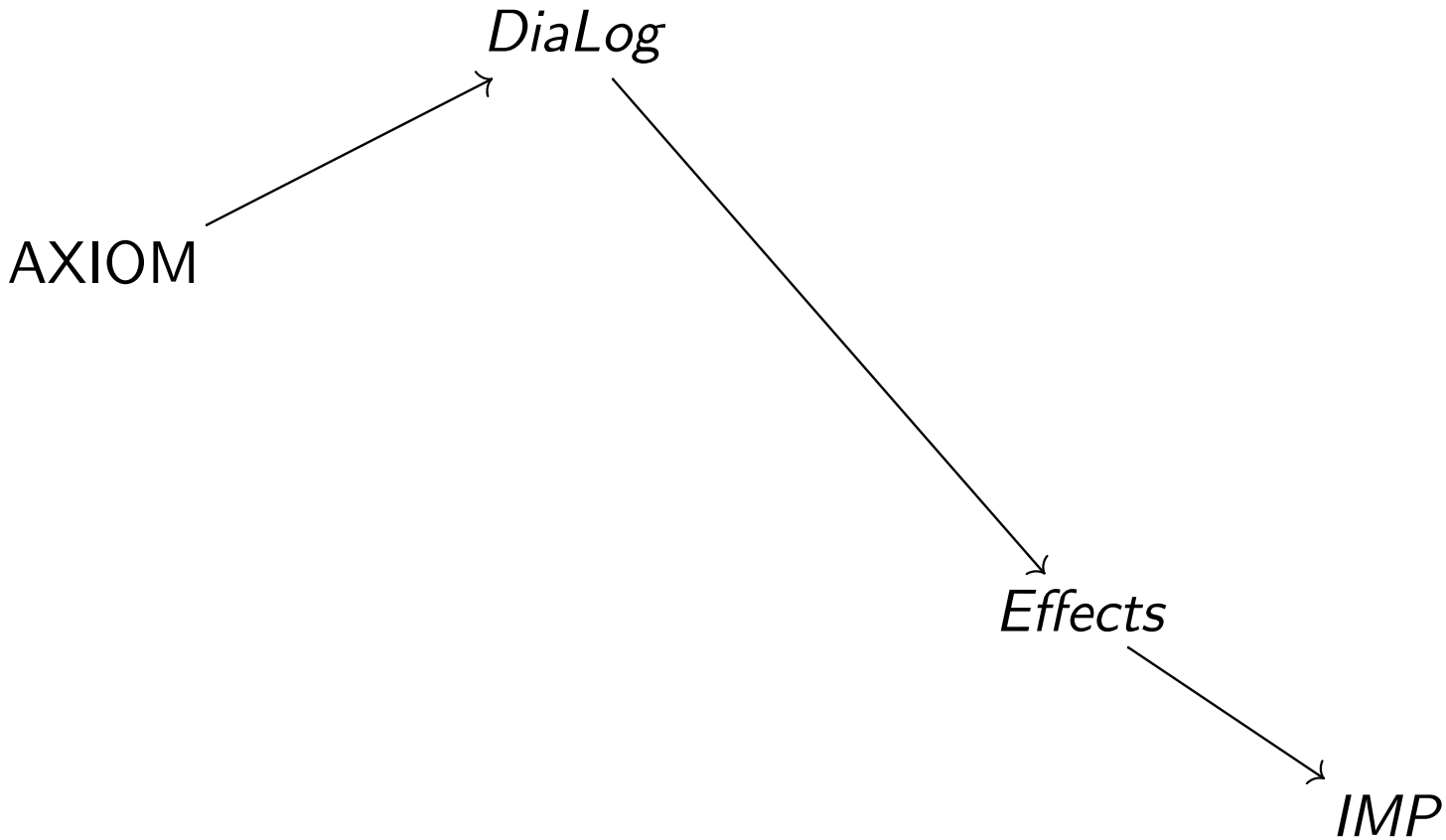
Dominique Duval

with the help of Jean-Guillaume Dumas, Burak Ekici,
Alexis Laouar, Damien Pous, Jean-Claude Reynaud

Algebraic Algorithms and Applications – Pisa – 31 March 2017

Happy birthday Patrizia!


AXIOM
IMP


DiaLog
AXIOM
Effects
IMP

DiaLog

$$\begin{array}{c} Th = F(Sp) \\ \Big\downarrow{\scriptstyle mod} \\ Dom \end{array}$$

Effects and IMP

$$\begin{array}{ccccc} Th & \longrightarrow & Th' & \longrightarrow & Th'' \ \cdots\cdots \\ \Big\downarrow{\scriptstyle mod} & & \Big\downarrow{\scriptstyle mod'} & & \Big\downarrow{\scriptstyle mod''} \\ Dom & \longrightarrow & Dom' & \longrightarrow & Dom'' \ \cdots\cdots \end{array}$$

Outline

From Axiom to DiaLog

AXIOM is (loosely) based on abstract data types (ADT)
and algebraic specifications (booleans, integers, lists, ...)
[developped by the ADJ group at IBM Research]

Question.
Can we find a more powerful, more accurate, theoretical basis?

- Institutions are too close to algebraic specifications
  [Goguen, Burstall]
- We have proposed the framework of Diagrammatic Logics
  [Domínguez, Duval, Lair]

"An inference rule is a (categorical) fraction"

The modus ponens rule

Written AS a fraction

$$\frac{A \quad A \Rightarrow B}{B}$$

"if A implies B and A is true, then B is true"

or in two steps:

"(A implies B and A is true) if and only if
(A implies B and A is true and B is true),
and [obviously] if (A implies B and A is true and B is true)
then (B is true)"

This rule IS a fraction

$$\{A,\ A \Rightarrow B\} \underset{\text{if and only if}}{\overset{\subseteq}{\longrightarrow \atop \leftarrow}} \{A,\ A \Rightarrow B,\ B\} \xleftarrow[\text{if then}]{\supseteq} \{B\}$$

Rules as fractions

A rule, written AS a fraction $\frac{H}{C}$, actually IS a fraction $\frac{c}{h}$

$$\frac{H}{C} \quad \text{or} \quad H \xrightarrow[\leftarrow - - -]{h} H' \xleftarrow{c} C \quad \text{or} \quad \frac{c}{h}$$

where $H' =$ "H and C", with respect to a functor $\mathbf{S} \xrightarrow{F} \mathbf{T}$

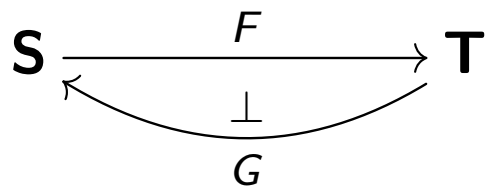
- Solid arrows $H \xrightarrow{h} H' \xleftarrow{c} C$ are in $\mathbf{S}$
- Dashed arrow $H \leftarrow - - H'$ stands for $F(H) \xleftarrow{F(h)^{-1}} F(H')$ in $\mathbf{T}$

– $\mathbf{S}$ is the category of specifications
– $\mathbf{T}$ is the category of theories
– $F(Sp)$ is the theory generated by the specification Sp

Logic as adjunction

Definition?

A diagrammatic logic is an adjunction $F \dashv G$ such that the counit $F \circ G \Rightarrow Id_{\mathbf{T}}$ is an iso, i.e., G is full and faithful



In addition, this adjunction must be "syntactic"

Definition!

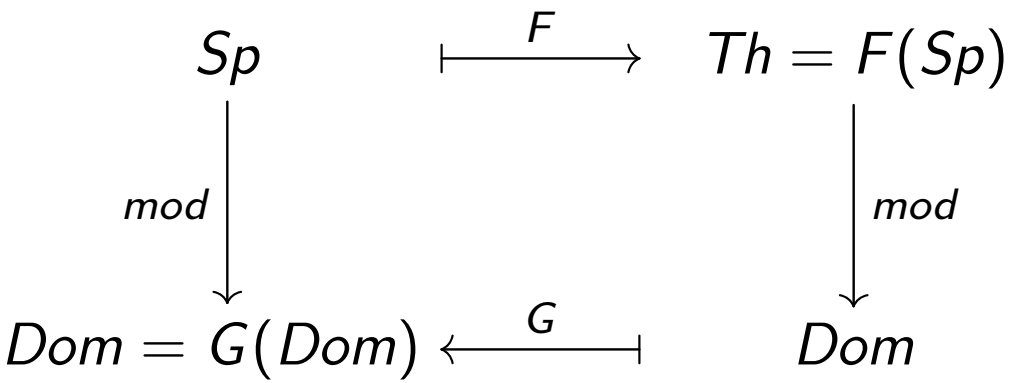
A diagrammatic logic is [determined by] a morphism of limit sketches which simply adds inverses to some arrows.

Models

Given a diagrammatic logic

A model of Th in Dom is a morphism $mod : Th \to Dom$ in $\mathbf{T}$

Thus, if $Th = F(Sp)$ (i.e., Th is presented by Sp) then
a model of Th in Dom is a morphism $mod : Sp \to Dom$ in $\mathbf{S}$



Morphisms as fractions

Given a diagrammatic logic

if $Th_1 = F(Sp_1)$ and $Th_2 = F(Sp_2)$ then
each morphism of theories $th : Th_1 \to Th_2$
is presented by a fraction

$$Sp_1 \xrightarrow{sp_1} Sp_2' \xleftarrow{sp_2} Sp_2$$

i.e., $th = F(sp_2)^{-1} \circ F(sp_1)$

$$Th_1 \xrightarrow{F(sp_1)} Th_2' \xrightarrow{F(sp_2)^{-1}} Th_2$$

Example: implementation of the operations in Sp_1
using the operations in Sp_2

Outline

From DiaLog to computational effects

There is a simple and powerful notion of morphism of diagrammatic logics.

This allows to deal with situations where the syntax and the semantics do not fit.

Example. In an imperative language with exceptions, a piece of program $p : x \to y$ is interpreted as a partial function $[\![p]\!] : S \times [\![x]\!] \rightharpoonup S \times [\![y]\!] + S \times E$

A computational effect involves several kinds of terms (values and computations, or pure and effectful) and here in addition

"A computational effect involves several kinds of equations"

State

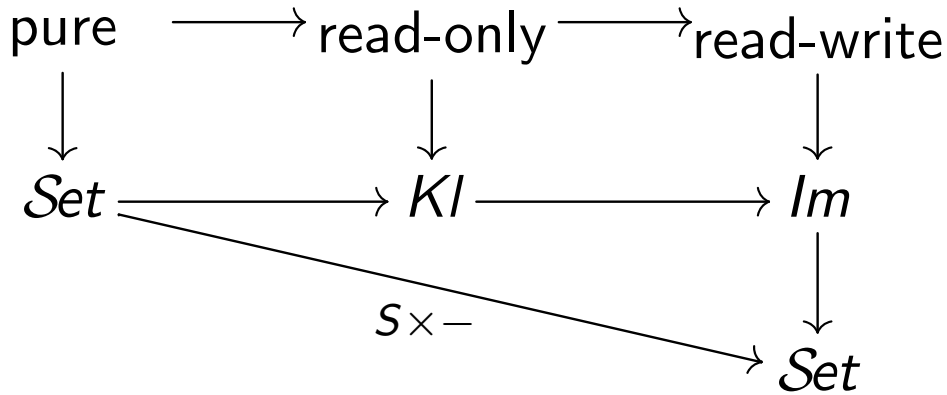
Our first motivation for building diagrammatic logic
was to get a proof system for programs involving states

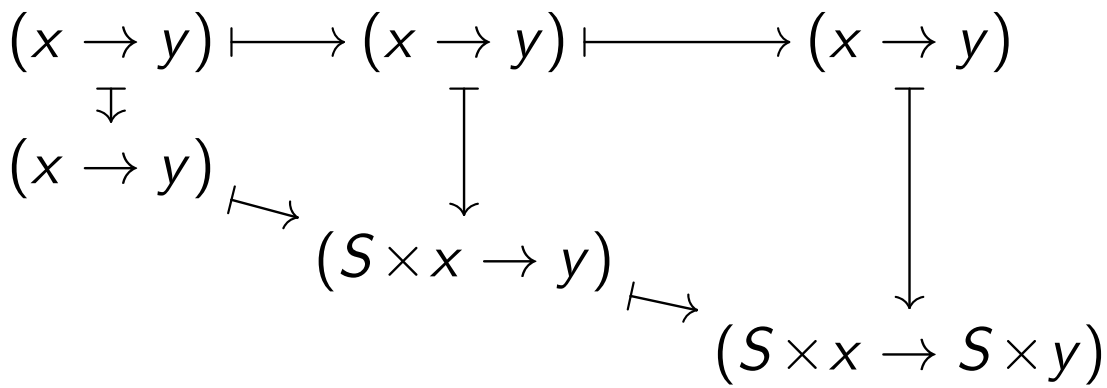
In an imperative language, we can distinguish 3 kinds of terms:
– pure terms
– accessors or read-only
– modifiers or read-write

A term $x \longrightarrow y$ in the syntax is interpreted
using the set S of states:
– pure: $x \longrightarrow y$
– read-only: $S \times x \longrightarrow y$
– read-write: $S \times x \longrightarrow S \times y$

Denotational semantics of states

Models in relevant logics involve the product comonad
$S \times - : \mathcal{S}et \to \mathcal{S}et$





"up-to-state" quasi-equations

The rules involve 2 kinds of "equations" on read-write terms:

– strong equations: $f_1 \equiv f_2 : x \to y$,
  interpreted as $f_1 = f_2 : S \times x \to S \times y$

– "up-to-state" quasi-equations: $f_1 \sim f_2 : x \to y$,
  interpreted as $pr \circ f_1 = pr \circ f_2 : S \times x \to y$

with different rules:

– strong equations form a congruence:
an equivalence relation compatible with composition:

$$\frac{g_1 \equiv g_2}{h \circ g_1 \circ f \equiv h \circ g_2 \circ f}$$

– "up-to-state" quasi-equations form a "weak" congruence:
an equivalence relation "weakly" compatible with composition:

$$\frac{g_1 \sim g_2}{h^{(pure)} \circ g_1 \circ f \sim h^{(pure)} \circ g_2 \circ f}$$

Operations on states

Let $Loc = \{X, Y, ...\}$ be the set of locations (or "variables")
(assumed of type integer Z)
– $lookup_X : \mathbb{1} \to Z$ is an accessor
– $update_X : Z \to \mathbb{1}$ is a modifier
Quasi-equations:

$$\begin{cases} lookup_X \circ update_X \sim id_Z \\ lookup_Y \circ update_X \sim lookup_Y \ \ (\text{if } Y \neq X) \end{cases}$$

Interpretation as required, when $S = \mathbb{Z}^{Loc} = \prod_{X \in Loc} \mathbb{Z}$
– $[\![lookup_X]\!] : S \to \mathbb{Z}$ such that $s \mapsto s(X)$
– $[\![update_X]\!] : S \times \mathbb{Z} \to S$ such that $(s, n) \mapsto s[n/X]$

States and exceptions: duality

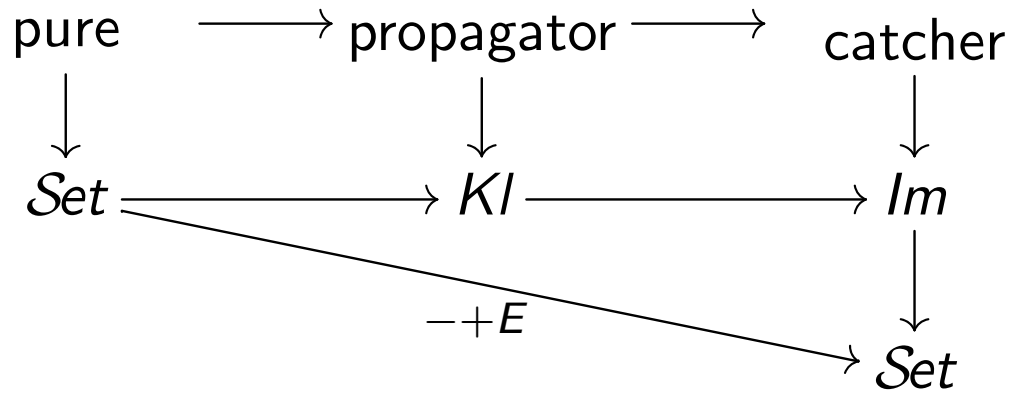
Then we realized that by duality from states
we get a proof system for programs involving exceptions

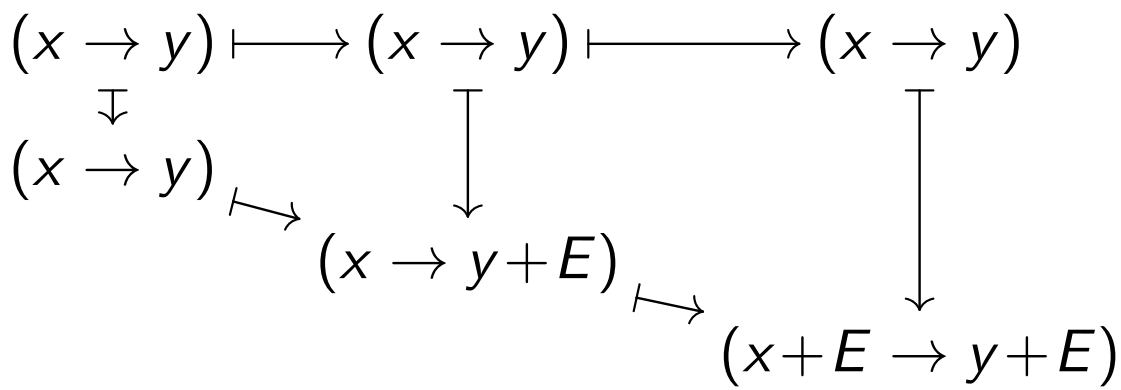
We distinguish 3 kinds of terms:
– pure terms
– propagators (that may throw and must propagate exceptions)
– catchers (that may recover from exceptions)

A term $x \longrightarrow y$ in the syntax is interpreted
using the set E of exceptions:
– pure: $x \longrightarrow y$
– propagator: $x \longrightarrow y + E$
– catcher: $x + E \longrightarrow y + E$

Denotational semantics of exceptions

Models in relevant logics involve the coproduct monad
$- + E : \mathcal{S}et \to \mathcal{S}et$





"up-to-exceptions" quasi-equations

The rules involve 2 kinds of "equations" on catchers:

– strong equations: $f_1 \equiv f_2 : x \to y$,
  interpreted as $f_1 = f_2 : x + E \to y + E$

– "up-to-exceptions" quasi-equations: $f_1 \sim f_2 : x \to y$,
  interpreted as $f_1 \circ in = f_2 \circ in : x \to y + E$

with different rules:

– strong equations form a congruence:
an equivalence relation compatible with composition:

$$\frac{g_1 \equiv g_2}{h \circ g_1 \circ f \equiv h \circ g_2 \circ f}$$

– "up-to-exceptions" quasi-equations form a "weak" congruence:
an equivalence relation "weakly" compatible with composition:

$$\frac{g_1 \sim g_2}{h \circ g_1 \circ f^{(pure)} \sim h \circ g_2 \circ f^{(pure)}}$$

Operations on exceptions

Let $Exc = \{e, e', ...\}$ be the set of exception names
(assumed with parameter of type integer Z)
– $tag_e : Z \to \mathbb{0}$ is a propagator
– $untag_e : \mathbb{0} \to Z$ is a catcher
Equations:

$$\begin{cases} untag_e \circ tag_e \sim id_Z \\ untag_e \circ tag_{e'} \sim tag_{e'} \quad (\text{if } e' \neq e) \end{cases}$$

Then tag_e and $untag_e$ have to be encapsulated for getting the required *throw* and *try/catch* constructions

What is a computational effect?

Effect = strong monad [Moggi]
Effect = Lawvere theory [Plotkin, Power, Hyland]
Effect = ?? I do not know...

Some features appear:

– several kinds of terms
– several kinds of "quasi-equations"

$$\begin{array}{ccccc} Th^{(0)} & \longrightarrow & Th^{(1)} & \longrightarrow & \dots \\ \downarrow{\scriptstyle mod^{(0)}} & & \downarrow{\scriptstyle mod^{(1)}} & & \\ Dom^{(0)} & \longrightarrow & Dom^{(1)} & \longrightarrow & \dots \end{array}$$

Combinaison of effects may look systematic by composition, but combinaison of quasi-equations is not systematic

Outline

From computational effects to IMP

Goal.

Design a proof assistant for imperative or object-oriented languages (based on Coq, for example)

– close to the syntax
– for proving equivalence of parts of programs

A case study.

The basic IMPerative language IMP: with the state effect

[and IMP-EX: with the state and the exceptions effects]

Actually, it is convenient to

"Consider conditionals and loops as effects"

IMP syntax

IMP is a very simple IMPerative language

$Loc = \{X, Y, ...\}$ is the set of locations (or "variables")

Expressions:

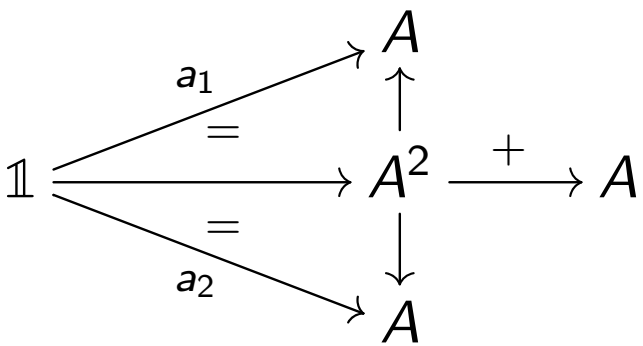
$$a ::= 0 \mid 1 \mid -1 \mid ... \mid X \mid Y \mid ... \mid a + a \mid ...$$
$$b ::= \mathit{true} \mid \mathit{false} \mid b \wedge b \mid ... \mid a = a \mid ...$$

Commands:

$$c ::= \texttt{skip} \mid c\,;c \mid X := a \mid \texttt{if } b \texttt{ then } c \texttt{ else } c \mid \texttt{while } b \texttt{ do } c$$

IMP syntax, categorically: expressions

- "types" A, B as objects,
- "type" *unit* or *void* as initial object $\mathbb{1}$
- expressions as arrows
- binary operations using products

EXPRESSION	a or b	$\mathbb{1} \xrightarrow{a} A$ or $\mathbb{1} \xrightarrow{b} B$
binary operation	$a_1 + a_2$	

IMP syntax, categorically: commands

- commands as arrows
- conditionals using coproducts

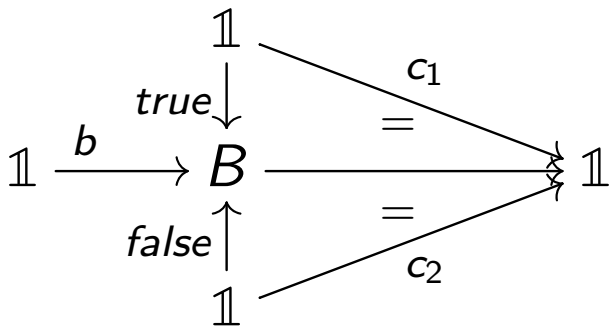
COMMAND	c
do-nothing	`skip`
sequence	$c_1; c_2$
assignment	$X := a$
conditional	`if` b `then` c_1 `else` c_2

$$\mathbb{1} \xrightarrow{c} \mathbb{1}$$

$$\mathbb{1} \xrightarrow{id} \mathbb{1}$$

$$\mathbb{1} \xrightarrow{c_1} \mathbb{1} \xrightarrow{c_2} \mathbb{1}$$

$$\mathbb{1} \xrightarrow{a} A \xrightarrow{X:=} \mathbb{1}$$



IMP denotational semantics

$S = \mathbb{Z}^{Loc} = \prod_{X \in Loc} \mathbb{Z}$ is the set of states

Expressions interpreted as total maps

$[\![a]\!] : S \to \mathbb{Z} = \{..., -1, 0, 1, ...\}$ e.g. $[\![X]\!](s) = s(X)$

$[\![b]\!] : S \to \mathbb{B} = \{true, false\}$

Commands interpreted as partial maps

$[\![c]\!] : S \rightharpoonup S$

$[\![X := a]\!](s) = s\,[[\![a]\!](s)/X]$

$[\![\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2]\!](s) = \mathit{if}\ [\![b]\!](s)\ \mathit{then}\ [\![c_1]\!](s)\ \mathit{else}\ [\![c_2]\!](s)$

$[\![\texttt{while } b \texttt{ do } c]\!] = \mathit{fix}(F_{[\![b]\!],[\![c]\!]})$

i.e., the least fixed-point of $F_{[\![b]\!],[\![c]\!]}$ where

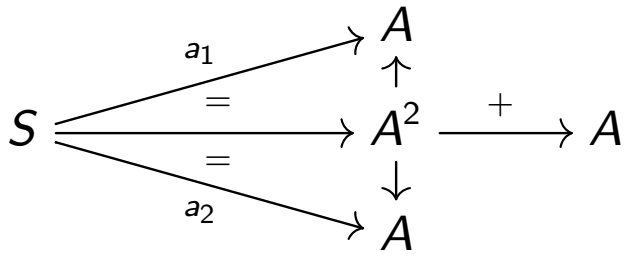
$(F_{[\![b]\!],[\![c]\!]}(f))(s) = \mathit{if}\ [\![b]\!](s)\ \mathit{then}\ f([\![c]\!](s))\ \mathit{else}\ s$

IMP denotational semantics, categorically

EXPRESSION

$S \xrightarrow{a} A$ or $S \xrightarrow{b} B$

binary operation



COMMAND

$S \xrightarrow{c} S$

do-nothing

$S \xrightarrow{id} S$

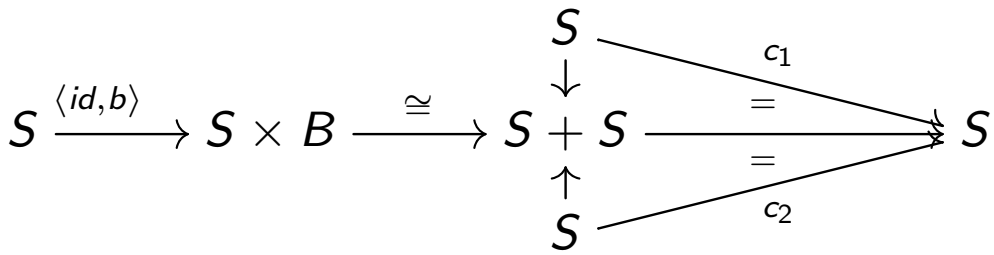
sequence

$S \xrightarrow{c_1} S \xrightarrow{c_2} S$

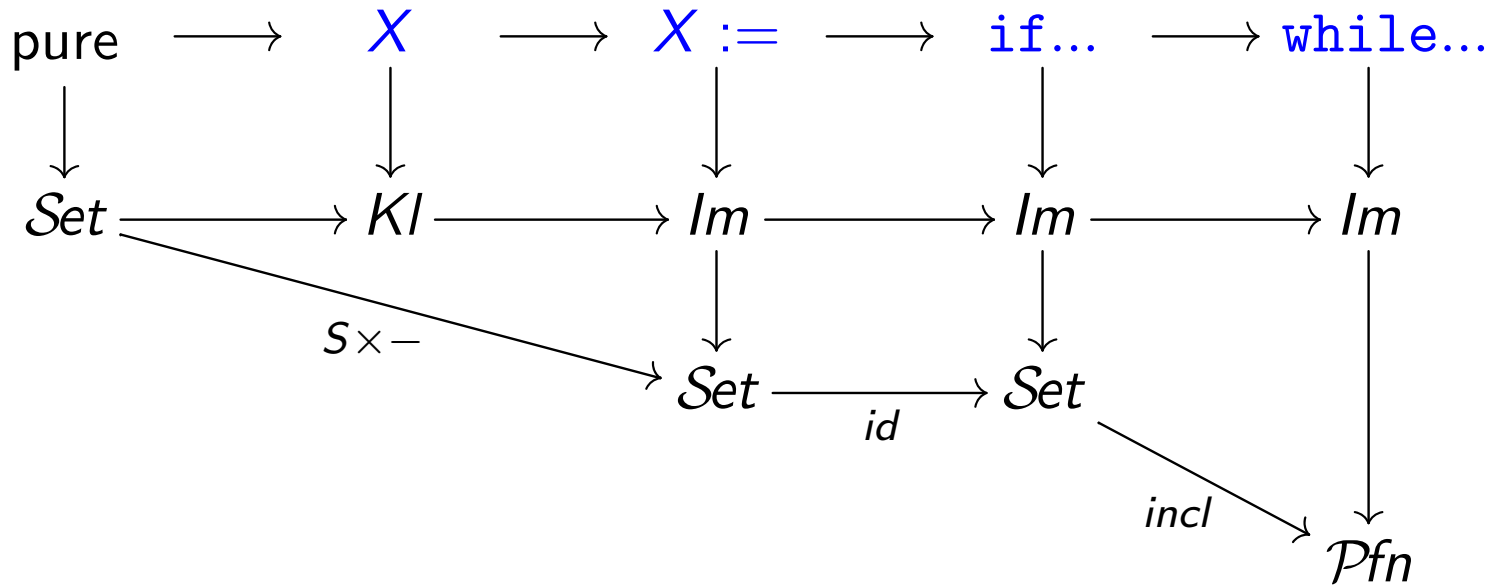
assignment

$S \xrightarrow{\langle id, a \rangle} S \times A \xrightarrow{X:=} S$

conditional



Effects in IMP



$$(x \to y) \quad (x \to y) \quad (x \to y) \quad (x \to y) \quad (x \to y)$$
$$\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow$$
$$(x \to y) \quad (S \times x \to y) \quad (S \times x \to S \times y) \quad (S \times x \to S \times y) \quad (S \times x \rightharpoonup S \times y)$$

Quasi-equations for IMP

Programs: $p ::= c\,;\, \texttt{return}\ (a)$

interpreted as $S \xrightarrow{\ c\ } S \xrightarrow{\ a\ } A$

– Quasi-equations for state: $p_1 \sim p_2 : \mathbb{1} \to A$

interpreted as $p_1 = p_2 : S \rightharpoonup A$

– Quasi-equations for conditionals: $c_1 \equiv_b c_2 : \mathbb{1} \to \mathbb{1}$,

interpreted as $c_1|_{S_b} = c_2|_{S_b} : S_b \rightharpoonup S$

where $S_b = \{s \in S \mid b(s) = \mathit{true}\} \subseteq S$

– Quasi-equations for loops: $c_1 \leq c_2 : \mathbb{1} \to \mathbb{1}$,

interpreted as $c_1 \leq c_2 : S \rightharpoonup S$ in $\mathcal{P}fn$

Combining quasi-equations

Example: combining $\sim$ (state) and $\leq$ (loop):

Quasi-equation $\preccurlyeq$ with $p_1 \preccurlyeq p_2 : \mathbb{1} \to A$
interpreted as $p_1 \leq p_2 : S \rightharpoonup A$

In particular:
if $p : \mathbb{1} \to A$ is a program and $r : \mathbb{1} \to A$ a pure expression, then

$$p \preccurlyeq r \iff r \text{ is the result of } p$$

Properties of quasi-equations

	$\equiv$	$\sim$	$\equiv_b$	$\leq$	$\asymp$
reflexive	V	V	V	V	V
transitive	V	V	V	V	V
symmetric	V	V	V	X	X
substitution	V	V	X	V	V
continuation	V	X	V	V	X

Conclusion

- categories of fractions for logic
- quasi-equations for computational effects
- conditionals and loops as effects for IMP

THANK YOU!