Semantics and Concurrence
Module on Semantic of Programming Languages

Pietro Di Gianantonio

Università di Udine

Presentation

Module of 6 credits (48 hours),

Semantics of programming languages
Describe the behavior of programs in a formal and rigorous way.

- We present 3 (2) different approaches, methods to semantics.
- Methods applied to a series of, more and more complex, programming languages.

Some topics overlapping the Formal Methods course.

Prerequisite for:

- Module of Concurrence (Lenisa),
- Abstract Interpretation (Comini),
- any formal and rigorous treatment of programs.

Semantics of programming languages

Objective: to formally describe the behavior of programs, and programming constructors.
As opposed to informal definitions, commonly used.

Useful for:

- avoid ambiguity in defining a programming language:
  - highlight the critical points of a programming language (e.g. static-dynamic environment, evaluation and parameter passing)
  - useful in building compilers,
- to reason about single programs (define logic, reasoning principles, programs):
  to prove that a given program to satisfy a give property, specification, that is correct.

Semantic approaches

- Operational Semantics, describes how programs are executed, not on a real calculator, too complex,
  on a simple formal machine (Turing machines style).
- Structured Operational Semantic (SOS). The formal machine consists of a rewriting system: system of rules, syntax.

Semantic approaches

- Denotational Semantics. The meaning of a program described by a mathematical object.
  Partial function, element in an ordered set.
  An element of an ordered set represents the meaning of a program, part of the program, program constructor.

  Alternative, action semantics, semantic games, categorical.
- Axiomatic Semantics. The meaning of a program expressed in terms of pre-conditions and post-conditions.

Several semantics because none completely satisfactory.
Each describes one aspect of program behavior, has a different goal.

SOS pros

Simple, syntactic, intuitive.

Quite flexible.

- It can easily handle complex programming languages.
- The structure of rules remains constant in different languages.

SOS cons

Semantics depend on syntax, it is formulated using syntax.

It is difficult to correlate programs written in different languages.

Semantics are not compositional (the semantics of an element depend on the semantics of its components).

It induces a notion of equivalence between programs difficult to verify (and use).

Denotational Semantics

Goals:

- a syntax-independent semantics, you can compare programs written in different languages.
- a compositional semantics (the semantics of an element depends on the semantics of its components).
- more abstract, provides tools for reasoning on programs.

Main feature: describe the behavior of a program through a mathematical object.

Different Features of Programming Languages

- no termination,
- store (memory, imperative languages)
- environment,
- not determinism,
- concurrence,
- higher order functions (functional languages),
- exceptions,
- ...

The complexity of denotation semantics grows rapidly, with the features of programming languages

Axiomatic semantics

Indirect description of a program,
through a set of assertions:

$$\{Pre\}\ p\ \{Post\}$$

It immediately gives a logic to reason about programs.

Complementary, and justified, by the other semantics.

Relations among semantics

- different descriptions of programming languages.
- one can prove the coherence among descriptions.
- one semantics descriptions can be justified by the others.

Textbook

Glynn Winskel:
The Formal Semantics of Programming Languages. An introduction.

A classic, simple, complete presentation, few frills, and few general considerations.

Three copies in the library.

We present a good part of the book, skipping:

- most of the proves of the theorems,
- the general introduction of formalism, the methodology: just presented on working examples.

Some additional topics.

Exam

- a set of exercise to solve at home,
- an oral exam, on appointment.

Prerequisites

Mathematical logic notions:

- calculation of predicates,
- set constructors: product, disjoint sum, function space, power set,
- grammar (free from context),
- inductive definition and induction principle,
- model theory: language and model, (syntax and semantics).

A simple imperative language: IMP

Syntactic categories:

Integer Numbers (**N**): n ; Boolean Values (**T**) b ; Locations (**Loc**): X.

Arithmetic Expressions (**AExp**):

$$a = n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$$

Boolean expressions (**BExp**):

$$b = \mathbf{true} \mid \mathbf{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \mathbf{not}\ b \mid b_0\ \mathbf{or}\ b_1 \mid b_0\ \mathbf{and}\ b_1$$

Commands (**Com**):

$$c ::= \mathbf{skip} \mid X := a \mid c_0 ; c_1 \mid \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1 \mid \mathbf{while}\ b\ \mathbf{do}\ c$$

IMP

Abstract syntax: ambiguous but simpler than concrete syntax

Incomplete syntax: unspecified syntax for numerical constants, locations:
not interesting, low level details, orthogonal to our discussion.

Minimum language capable of computing all computable functions (Turing-complete), if the locations can store arbitrarily large spaces.

Missing:

- variables (environment),
- procedure, function definitions,
- recursive definitions.

Operational Semantic for IMP

A set of rules to describe the behavior of arithmetic and Boolean expressions, commands.

One associates to arithmetic expressions assertions, judgments in the form:

$$\langle a, \sigma \rangle \;\Rightarrow\; n$$

where $\sigma : \mathbf{Loc} \rightarrow \mathbf{N}$, state (memory, store).

Derived judgments by rules in natural deduction,
Rules driven by syntax (structured operational semantics).

Axiom for the basic expressions:

$$\frac{}{\langle n, \sigma \rangle \;\Rightarrow\; n}$$

$$\frac{}{\langle X, \sigma \rangle \;\Rightarrow\; \sigma(X)}$$

IMP: SOS rules

To composite derivative rules:

$$\frac{\langle a_0, \sigma\rangle \;\Rightarrow\; n_0 \qquad \langle a_1, \sigma\rangle \;\Rightarrow\; n_1}{\langle a_0 + a_1, \sigma\rangle \;\Rightarrow\; n} \qquad n_0 + n_1 = n$$

...

- Evaluating an arithmetic expression is trivial, so trivial rules.
- Each expression has associated one rule, determined by its main connective.
- From rules one can derive a deterministic algorithm for evaluating an expression,
  (easily definable via pattern matching).

SOS rules

- Rules assume, via side condition, a preexisting mechanism for calculating the arithmetic operation.
  One abstracts from the problem of implementing the arithmetic operations:
  - chose a representation for number,
  - define algorithms for arithmetic operations
  - feasible via SOS rules,
  - one avoid to deal with these low level aspects
- Similarly, rules do not defined the syntax for store and a to perform operation on stores.
  One abstracts from the problem of implementing store operations.

Exercise: evaluate $2 * x - 3 + y$ in a store σ, $\sigma(x) = 4, \sigma(y) = 1$

Boolean expression: SOS rules

Axioms . . .

$$\frac{\langle a_0, \sigma\rangle \;\Rightarrow\; n \qquad \langle a_1, \sigma\rangle \;\Rightarrow\; n}{\langle a_0 = a_1, \sigma\rangle \;\Rightarrow\; \mathbf{true}}$$

$$\frac{\langle a_0, \sigma\rangle \;\Rightarrow\; n \qquad \langle a_1, \sigma\rangle \;\Rightarrow\; m}{\langle a_0 = a_1, \sigma\rangle \;\Rightarrow\; \mathbf{false}} \qquad n \neq m$$

. . .

$$\frac{\langle b_0, \sigma\rangle \;\Rightarrow\; t_0 \qquad \langle b_1, \sigma\rangle \;\Rightarrow\; t_1}{\langle b_0 \;\mathbf{and}\; b_1, \sigma\rangle \;\Rightarrow\; \mathbf{t}}$$

where $t \equiv$ **true** if $t_0 \equiv$ **true** and $t_1 \equiv$ **true**. Otherwise $t \equiv$ **false**.

Boolean operators

.

- Boolean operators can be evaluated in different ways: short circuit evaluations.
- This alternative evaluation can be express by the SOS rules.
- Four alternative rules for connectivity **and** .

Exercises

Through the rules one can explicitly define the arithmetic operation, without using side conditions.
Simplification: consider the Peano natural numbers and not the integers, i.e. the grammar:

$$n ::= 0 \mid Sn$$

Addition, product, comparison rules.

The same problem with binary notation:

$$n = 0 \mid n : 0 \mid n : 1$$

Where $[\![n : 0]\!] = 2 \times [\![n]\!]$
e $[\![n : 1]\!] = 2 \times [\![n]\!] + 1$

Commands

Executing a command has the effect of modifying memory, store.
Judgements have form:

$$\langle c, \sigma \rangle \;\Rightarrow\; \sigma'$$

To represent updated store, one uses the notation $\sigma[m/X]$

$$\begin{aligned} &\sigma[m/x](X) = m \\ &\sigma[m/X](Y) = \sigma(Y) \qquad \text{if } X \neq Y \end{aligned}$$

In a complete operational approach the state should be a syntactic object:

- grammar to define states: ground states, updating operations,
- set of rules describing the behavior.

Commands, rules

$$\langle \mathbf{skip}, \sigma \rangle \;\Rightarrow\; \sigma$$

$$\frac{\langle a, \sigma \rangle \;\Rightarrow\; n}{\langle X := a,\; \sigma \rangle \;\Rightarrow\; \sigma[n/x]}$$

$$\frac{\langle c_0, \sigma \rangle \;\Rightarrow\; \sigma' \qquad \langle c_1, \sigma' \rangle \;\Rightarrow\; \sigma''}{\langle c_0; c_1,\; \sigma \rangle \;\Rightarrow\; \sigma''}$$

$$\frac{\langle b, \sigma \rangle \;\Rightarrow\; \mathbf{true} \qquad \langle c_0, \sigma \rangle \;\Rightarrow\; \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1,\; \sigma \rangle \;\Rightarrow\; \sigma'}$$

$$\ldots$$

$$\frac{\langle b, \sigma \rangle \;\Rightarrow\; \mathbf{true} \qquad \langle c, \sigma \rangle \;\Rightarrow\; \sigma' \qquad \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma' \rangle \;\Rightarrow\; \sigma''}{\langle \mathbf{while}\ b\ \mathbf{do}\ c,\; \sigma \rangle \;\Rightarrow\; \sigma''}$$

$$\ldots$$

Equivalences:

Semantics induces a notion of equivalence between commands (and expressions):

$$c_0 \sim c_1$$

if for every pair of stores σ, σ':

$$\langle c_0, \sigma \rangle \ \Rightarrow \ \sigma' \quad \text{if and only if} \quad \langle c_1, \sigma \rangle \ \Rightarrow \ \sigma'$$

Several notions of equivalence are possible on the same structure:

- $$c_0 \equiv c_1$$

  if the c_0 and c_1 commands, formatted in the abstract syntax, are the same.
- On stores, define as syntactic objects:
  - σ_0 and σ_1 are the same syntactic object;
  - σ_0 and σ_1 define the same function $\mathbf{Loc} \to \mathbf{N}$.

Exercises

- Encode in Haskell the rules of operational semantics. Mostly straightforward, the only difficulty is the encoding of the store.
- Given:

$$w \equiv \textbf{while } b \textbf{ do } c$$

show that:

$$w \sim \textbf{if } b \textbf{ then } c; w \textbf{ else skip}$$

and

$$w \sim \textbf{if } b \textbf{ then } w \textbf{ else skip}$$

Metatheory: determinism

The rules are deterministic:

- Weak formulation:
  For every c, σ exists a single σ' for which it is valid

$$\langle c, \sigma \rangle \;\Rightarrow\; \sigma'$$

- Strong formulation:
  For each c, σ exists a single σ' and a single demonstration of:

$$\langle c, \sigma \rangle \;\Rightarrow\; \sigma'$$

Big-step, small-step SOS

Alternative formulation: describes a step of computation.

$$\langle c, \sigma \rangle \ \rightarrow \ \langle c', \sigma' \rangle$$

New rules for the while

$$\frac{\langle b, \sigma \rangle \ \rightarrow \ \textbf{true}}{\langle \textbf{while } b \textbf{ do } c, \ \sigma \rangle \ \rightarrow \ \langle c; \textbf{while } b \textbf{ do } c, \ \sigma \rangle}$$

Second rule for the while . . .

$$\langle a, \sigma \rangle \ \rightarrow \ \langle a', \sigma' \rangle$$

One can define the evaluation order of the arguments

$$\frac{\langle a_0, \ \sigma \rangle \ \rightarrow \ \langle a_0', \ \sigma \rangle}{\langle a_0 + a_1, \ \sigma \rangle \ \rightarrow \ \langle a_0' + a_1, \ \sigma \rangle}$$

$$\frac{\langle a_1, \ \sigma \rangle \ \rightarrow \ \langle a_1', \ \sigma \rangle}{\langle n + a_1, \ \sigma \rangle \ \rightarrow \ \langle n + a_1, \ \sigma \rangle}$$

Big-step, small-step SOS

Both formulations are used:

- For some languages it is easier to provide big-step semantics. More abstract.
- Small-step SOS provide more detail on how the computation is performed
  Can be used analyzed the complexity of algorithm, provided that single step are really an elementary step of computation.
- In the languages for concurrency it is crucial to consider the small-step SOS.
  Contains additional information about computing steps and execution order.
- It can be not trivial to demonstrate the equivalence between the small-step and big-step description of the same language.

Induction

In mathematics and computer science many sets are define inductively:

- natural numbers,
- lists,
- grammars,
- derivations, demonstrations

Each inductive definition is characterize by a set constructors:

- zero, successor;
- empty list, concatenation;
- constructs of grammar;
- axioms, derivation rules.

Induction

On inductive sets one can:

- define functions by recursion: recursive definitions,
- prove theorems by induction: inductive demonstrations.

It is sometime convenient to use stronger inductive principle (recursive definition).
For example:

- generalized induction on natural number. If

$$\forall n \,.\, (\forall m < n \,.\, P(m)) \Rightarrow P(n)$$

  then

$$\forall n \,.\, P(n)$$

These generalizations are defined in terms well-founded sets.

Denotational Semantics for IMP

associates, to IMP expressions, mathematical objects: (partial) functions,
is compositional.

Building basics:

- $N = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$, the set of integers.
- $T = \{true, false\}$, the set of Boolean values.
- $\Sigma = \mathbf{Loc} \rightarrow N$, the set of possible states (memory, store configurations).

Note the difference between $\mathbf{N}$ and N.

Interpreting Functions

Each syntactic category is associated with an interpretation function:

- $\mathcal{A}[\![\ ]\!] : \textbf{AExp} \ \rightarrow (\Sigma \rightarrow N)$
  an arithmetic expression represents a function from integer state.
- $\mathcal{B}[\![\ ]\!] : \textbf{BExp} \ \rightarrow (\Sigma \rightarrow T)$
- $\mathcal{C}[\![\ ]\!] : \textbf{Com} \ \rightarrow (\Sigma \rightharpoonup \Sigma)$
  a command represents a partial function, from state to state.

Resuming the ideas of operational semantics, but expressing them differently.

The double parenthesis $[\![\ ]\!]$ are used to enclose syntactic elements

Semantics of expressions (arithmetic, Boolean)

Interpretation functions are defined by induction on the grammar (on the structure of the term).

$$\mathcal{A}[\![\mathbf{n}]\!](\sigma) = n$$

$$\mathcal{A}[\![\mathbf{X}]\!](\sigma) = \sigma(X)$$

$$\mathcal{A}[\![a_o \mathbf{+} a_1]\!](\sigma) = (\mathcal{A}[\![a_o]\!](\sigma)) + (\mathcal{A}[\![a_1]\!](\sigma))$$

...

$$\mathcal{B}[\![a_o \mathbf{\leq} a_1]\!](\sigma) = (\mathcal{A}[\![a_o]\!](\sigma)) \leq (\mathcal{A}[\![a_1]\!](\sigma))$$

Each element, each operator, is interpreted with its semantic correspondent.

Commands semantics

Commands represent partial functions.

In the definition, partial functions
are seen as relations (the set of pairs " argument, value ")
represented through their graphs.

$$\mathcal{C}[\![\textbf{skip}]\!] = \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$$

$$\mathcal{C}[\![X := a]\!] = \{(\sigma, \sigma[n/X]) \mid \sigma \in \Sigma, \quad \mathcal{A}[\![a]\!](\sigma) = n\}$$

$$\mathcal{C}[\![c_0; c_1]\!] = \mathcal{C}[\![c_1]\!] \circ \mathcal{C}[\![c_0]\!]$$

$$\begin{aligned}\mathcal{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!] = \quad & \{(\sigma, \sigma') \in \mathcal{C}[\![c_0]\!] \mid \mathcal{B}[\![b]\!](\sigma) = \mathit{true}\} \ \cup \\ & \{(\sigma, \sigma') \in \mathcal{C}[\![c_1]\!] \mid \mathcal{B}[\![b]\!](\sigma) = \mathit{false}\}\end{aligned}$$

The hard case: **while** constructor

$$\begin{array}{l}\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] = \mathcal{C}[\![\textbf{if } b \textbf{ then } c; \textbf{while } b \textbf{ do } c \textbf{ else skip}]\!] = \\ \{(\sigma,\sigma) \mid \sigma \in \Sigma,\ \mathcal{B}[\![b]\!](\sigma) = \mathit{false}\} \ \cup \\ \{(\sigma,\sigma') \in (\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] \circ \mathcal{C}[\![c]\!]) \mid \mathcal{B}[\![b]\!](\sigma) = \mathit{true}\}\end{array}$$

recursive definition.

Existence of solution:

- reduce the problem to a fixed-point problem;
- consider the operator:

$$\begin{array}{rl}\Gamma(R) = & \{(\sigma,\sigma) \mid \sigma \in \Sigma,\ \ \mathcal{B}[\![b]\!](\sigma) = \mathit{false}\} \cup \\ & \{(\sigma,\sigma') \in (R \circ \mathcal{C}[\![c]\!]) \mid \mathcal{B}[\![b]\!](\sigma) = \mathit{true}\}\end{array}$$

  Γ transforms a relation between Σ and Σ into another relation.

$$\Gamma : \mathit{Rel}(\Sigma,\Sigma) \ \rightarrow\ \mathit{Rel}(\Sigma,\Sigma)$$

- define $\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]$ as minimum fixed-point for Γ.

Background Idea

Given

$$\Omega \equiv \textbf{while true do skip}$$

Define

$$\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]$$

As limit of its approximations:

$$\mathcal{C}[\![\Omega]\!]$$

$$\mathcal{C}[\![\textbf{if } b \textbf{ then } c; \Omega \textbf{ else skip}]\!]$$

$$\mathcal{C}[\![\textbf{if } b \textbf{ then } c; (\textbf{if } b \textbf{ then } c; \Omega \textbf{ else skip}) \textbf{ else skip}]\!]$$

Fixed-point theorems (on orders)

They provide solution to the previous problem.

- Knaster-Tarski theorem: in a complete lattice, each monotone function has a minimum (and a maximum) fixed-point, (important in math)
- in a complete partial order each monotone and continuous function has a minimum fixed-point.
  (fundamental in denotational semantics)
  (the property of the order weakens, the properties of the function are strengthened).
- in a complete metric space, each contractive function has a single fixed-point,
  (used in analysis: each contractive function on reals has a single fixed-point)

Ordered set

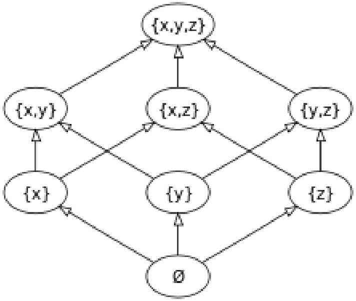
Definition (Partial order)

A partial order $(P, \sqsubseteq)$ is composed by a set P and a binary relation $\sqsubseteq$ on P s.t. for every $p, q, r \in P$

- $p \sqsubseteq p$, (reflexive)
- $p \sqsubseteq q$ and $q \sqsubseteq r$ then $p \sqsubseteq r$, (transitive)
- $p \sqsubseteq q$ and $q \sqsubseteq p$ then $p = q$, (antisymmetric)

Examples

- natural numbers, rational, reals, (total order);
- powerset, finite powerset (Boolean algebra);
- partial functions ($Bool \rightharpoonup 0, \quad Nat \rightharpoonup Nat$)



Least upper bound

Definition

Given subset of X of P,

- p is an upper bound for X if $\forall q \in X\ .\ q \sqsubseteq p$.
- The least upper bound (l.u.b) of X, $\bigsqcup X$ if it exists, is the smallest among the upper bounds, that is:
    - $\forall q \in X\ .\ q \sqsubseteq \bigsqcup X$, and
    - for each p, if $\forall q \in X\ .\ q \sqsubseteq p$ then $\bigsqcup X \ \sqsubseteq\ p$

  the least upper bound is unique (show by exercise).
- the dual notions are:
    - lower bound
    - greatest lower bound (g.l.b), $\bigsqcap$

Least upper bound

In an powerset,
l.u.b coincides with union,
g.l.b coincide with intersection.

In a linear order,
l.u.b, of a finite set, is the maximum
g.l.b, of a finite set, is the minimum.

Lattice

Definition (Lattice)

- A lattice is a partial order in which each pair of elements has an least upper bound and a greatest lower bound end of $\sqcap$.
  We can write $\bigsqcup\{p, q\}$ as $p \sqcup q$.
  It follows that in a lattice every finished set has top set.
- A complete lattice is a partial order in which each subset has an least upper bound.

Exercise: show that every subset in a complete lattice also has a greatest lower bound.

Knaster-Tarski Theorem

Definition (Monotone)

A function $f : P \to Q$ between orders is monotone if it respects the order.
If $p_1 \sqsubseteq p_2$ then $f(p_1) \sqsubseteq f(p_2)$

Theorem (Knaster-Tarski)

Each monotone function f on a complete lattice P has a minimum (maximum) fixed-point.

Proof.

One can show that $\bigsqcap\{p \mid f(p) \sqsubseteq p\}$ is a fixed-point for f (i.e. $\bigsqcap\{p \mid f(p) \sqsubseteq p\} = f(\bigsqcap\{p \mid f(p) \sqsubseteq p\})$ $\square$

Relations

The partial functions do not form a lattice.
Relations are an extension of the partial functions and form a lattice.
Partial functions and functions can be seen as special cases of relations.

Definition (Relations and functions)

- A relation between X and Y is a subset of $X \times Y$.
- A partial function $f : X \rightharpoonup Y$ is a relation such that $\forall x \in X, y, y' \in Y \;.\; (x, y) \in f \;\rightarrow\; y = y'$
- a total function is a partial function such that $\forall x \exists y \;.\; (x, y) \in f$

$(x, y) \in f$ can also be written by $y = f(x)$

Relations

Definition (Composition)

Given two R relations between X and Y and S between Y and Z we define $S \circ R$ as

$$(x, z) \in S \circ R \Leftrightarrow \exists y \;.\; (x, y) \in R \wedge (y, z) \in S$$

Note the inversion in the order.

If relations are also functions: composition as relation coincides with the composition as function.

Application to semantics

- the set of relations on $\Sigma \times \Sigma$ forms a complete lattice (actually a complete boolean algebra),
- the operator:

$$\begin{aligned}\Gamma(R) = \ & \{(\sigma,\sigma) \mid \mathcal{B}[\![b]\!](\sigma) = \mathit{false}\} \cup \\ & \{(\sigma,\sigma') \in (R \circ \mathcal{C}[\![c]\!]) \mid \mathcal{B}[\![b]\!](\sigma) = \mathit{true}\}\end{aligned}$$

  is monotone;
- the Knaster-Tarski theorem proves the existence minimum fixed-point for the Γ operator (recursive definition of semantics).

Recursive definition has multiple solutions, the minimum solution is the one that correctly describes the behavior of the program.

Structures for denotational semantics

- It is preferable to use partial functions instead of relations. The previous solution does not prove that the fixed-point is a partial function, it could be a generic relation.
- Remind: in IMP, the semantics of a command: $\Sigma \rightharpoonup \Sigma$.
- The set of partial functions forms an order,
  $f \sqsubseteq g$ when the following equivalent conditions are met:
  - g is more defined than f,
  - $\forall x . f(x) \downarrow \Rightarrow f(x) = g(x)$
  - as a set of pairs (argument, result) $f \subseteq g$

  this space is not a lattice.
- It is necessary to use a second fixed-point theorem.

Complete Partial Orders

Definition (CPO)

- In a partial order P, an ω-chain (a chain having length ω) is countable sequence of elements in P with each element greater (or equal) than the previous one

$$p_1 \sqsubseteq p_2 \sqsubseteq p_3, \ldots$$

- A complete partial order (CPO) P is a partial order in which each ω-chain has a least upper bound.
- A complete partial order with bottom is a complete partial order containing a minimum element $\perp$.

CPO are the typical structures to interpret programs.

An example of CPO with bottom is the set of partial functions from any pair of sets (from N to N).

Continuous functions on CPO

Definition (Continuity)

A function $f : D \to E$ between CPO is continue if it preserves the least upper bound of of the chains (the limit of chains)
For any chain $p_1, p_2, p_3, \ldots$:

$$\bigsqcup_{i \in N} f(p_i) = f(\bigsqcup_{i \in N} p_i).$$

Similarly, in mathematical analysis a function is continuous iff preserves limits of convergent sequences.

New fixed-point theorem

Theorem

Each function continues on a CPO with bottom $f : D \to D$ has a minimum fixed-point.

Proof sketch.

The minimum fixed-point is the upper end of the chain
$\bot,\ f(\bot),\ f(f(\bot)),\ f^3(\bot), \ldots$ $\square$

Notice that the proof is constructive, suggest a way to compute the fixed-point.

The proof of Knaster-Tarski is not constructive, use glb of uncountable subsets.

Semantics of **while**

Remind that the function:

$$\mathbf{C}[\![\textbf{while } b \textbf{ do } c\ ]\!]$$

is defined as the minimum fixed point of Γ

$$\begin{aligned}\Gamma(R) = \ & \{(\sigma,\sigma) \mid \sigma \in \Sigma,\ \ \mathcal{B}[\![b]\!](\sigma) = \mathit{false}\} \cup \\ & \{(\sigma,\sigma') \in (R \circ \mathcal{C}[\![c]\!]) \mid \mathcal{B}[\![b]\!](\sigma) = \mathit{true}\}\end{aligned}$$

One should prove that Γ is monotone (easy), and continuous (non easy, we postpone the proof).

Definition of **while** by approximations

By the proof of fixed-point theorem, the function

$$\mathbf{C}[\![\textbf{while } b \textbf{ do } c\ ]\!]$$

the limit of the following ω-chain of partial functions:

$$\begin{array}{l}
\mathcal{C}[\![\Omega]\!] \\
\mathcal{C}[\![\textbf{if } b \textbf{ then } (c; \Omega)]\!] \\
\mathcal{C}[\![\textbf{if } b \textbf{ then } (c; \textbf{if } b \textbf{ then } (c; \Omega))]\!] \\
\mathcal{C}[\![\textbf{if } b \textbf{ then } (c; \textbf{if } b \textbf{ then } (c; \textbf{if } b \textbf{ then } (c; \bot)))]\!] \\
\ldots
\end{array}$$

where Ω stands for an always divergent command
while true do skip
and **if** b **then** c stands for the command: **if** b **then** c **else skip**.

Syntactic sugar, enriches the language without adding new semantic definitions.

Examples

Compute:

$$\mathbf{C}[\![\textbf{while true do skip}\,]\!];$$

the operational and denotational semantics of

$$\textbf{while } 2 \leq X \textbf{ do } X := X - 2$$

and

$$\begin{array}{l} Z := 0; \\ \textbf{while } Y \leq X \textbf{ do} \quad \begin{array}{l} X := X - Y; \\ Z := Z + 1 \end{array} \end{array}$$

Store can be limited to the locations contained in the command.

Operational and denotational semantics agree

Theorem

The judgement

$$\langle c, \sigma \rangle \;\Rightarrow\; \sigma'$$

is derivable iff

$$(\sigma, \sigma') \in \mathcal{C}[\![c]\!]$$

sketch.

By induction on the syntax of c, and
for the special case where c is a **while** command, by inductions on the derivation. □

Domain theory

Mathematical foundation for denotational semantics:

- describe the structure used in denotational semantics,
- enumerate their properties

Generalizing the previous example (denotational semantics of IMP) programming languages (and their components: data types, commands) are interpreted using:

1. complete partial orders (CPOs)
    - order: the information order;
    - $\perp$ represents the absence of information, the program that always diverges,
    - allows solutions to recursive definitions, fixed-point equations.
2. functions between CPOs that are
    - monotonic
    - continue: preserve the least upper bounds of ω-chains.

Examples of CPO

- N with flat order: $n \sqsubseteq m \;\Leftrightarrow\; n = m$.
- $N_\perp = N \cup \{\perp\}$, $n \sqsubseteq m \;\Leftrightarrow\; (n = m \;\vee\; n = \perp)$ (the set of natural number with $\perp$),
- $T = \{\mathbf{true}, \mathbf{false}\}$, $T_\perp$
- $\mathbf{O} = \{\perp, \top\}$, with $\perp \sqsubseteq \top$ (Sierpinski space).
- $N \to N_\perp$ with the point-wise order:
  - $f \sqsubseteq g$ iff for every n, $f(n) \sqsubseteq g(n)$,
  - isomorphic to $N \rightharpoonup N$ (with the order defined above)
  - internalize the divergence: $f(n) = \perp$ denotes the divergence of $f(n)$,
  - partial functions are avoided

Partial order that are not CPO

- natural number with standard order
- finite powerset of natural numbers

Examples of CPO

- $\Omega =$ natural numbers $+ \{\infty\}$, with the linear order.
- Powerset
- the lazy natural numbers,
  Haskell computation having type `data Nat = Zero | Suc Nat`
- streams of Boolean values: partial strings, arbitrarily long.
  `data StreamBool = TT StreamBool | FF StreamBool`

Monotonicity

Intuitive meaning.

Consider a program with a functional type
$F : (Nat \rightharpoonup Bool) \rightharpoonup Bool$

monotone, if $F(f) \Rightarrow$ **true** then,
for any function g more define that f, $F(g) \Rightarrow$ **true** .

Preserves the information order: more information in input generate more information in output.

Continuity

A functional type $F : (Nat \rightharpoonup Bool) \rightharpoonup Bool$ is

- continuous if $F(f) \Rightarrow \mathbf{true}$ then F generates **true** after evaluating f on a finite number of values,
- i.e., there must exists a partial function g, defined on a finite number of elements, such that $g \sqsubseteq [\![f]\!]$ and $[\![F]\!](g) = \mathbf{true}$
- functions need to be finitary: To generate a finite piece of information as the result, computation examine just a finite piece of information of the argument.

Example of non continuous functionals:

- infinitary and: test if a function return true on any input,
- test if a function, on integers, converges on even numbers.

Exercise: show that composition preserves continuity.

Lambda (λ) notation

In mathematics, one defines functions by means of equations:
$f(x) = \sin(x) + \cos(x)$.

With the λ notation one write directly:
$\lambda x \ . \ \sin(x) + \cos(x)$, or $\lambda x \in R \ . \ \sin(x) + \cos(x)$
or $f = \lambda x. \sin(x) + \cos(x)$

In Haskell:

```
\x -> (sin x) + (cos x)
```

Benefits

- name less functions, one can define a function without giving it a name,
- definition of functions more synthetic,
- functions similar to other elements, first class objects
- conceptually clearer: $\int \sin(x) dx$ becomes $\int \lambda x. \sin(x)$ or $\int \sin$.

CPO Builders

To give semantics to complex languages, one associates to

- types: suitable structured CPOs;
- programs and program subexpressions: CPO elements, CPO functions

To do this:

- new CPO are build from existing ones,
- standard operators and functions are used in the process.

Discrete CPO

To a set of D values (ex. N the integer set, B),
one associates a CPO D with flat order $d_1 \sqsubseteq d_2$ iff $d_1 = d_2$.

With respect to the information order, all elements are unrelated:

- different information,
- all elements completely defined,
- none more defined than the other.

A set of completely defined values.

CPO without bottom.

Lifting

Definition

$D_\perp = D \cup \{\perp\}$ with order relation:
$d_1 \sqsubseteq d_2$ iff $d_1 = \perp$ or $(d_1, d_2 \in D \ \wedge \ d_1 \sqsubseteq d_2)$

One add to D, an element $\perp$ representing a divergent computations, the other elements represents computation generating a D elements.

Associated functions:

- $\lfloor \_ \rfloor : D \to D_\perp$,
  given a value d, constructs a computation $\lfloor d \rfloor$ returning d as result.
- from a function $f : D \to E$, defined on values,
  construct a function $f^* : D_\perp \to E$ defined on computations,
  E need to be a CPO with bottom.

Notation. $(\lambda x.e)^*(d)$ is also written as *let* $x \Leftarrow d \ . \ e$.

Some remarks

- we use an approach inspired by category theory:
  - for each constructor, we defined the set of functions characterizing it,
  - these functions are constructors or destructors
  - basic ingredients for building other functions,
- CPO builders have a corresponding type constructors in Haskell,
- in category theory the operation of Lifting defines a monad, correspondent Haskell: the Monad Maybe
  unmatched correspondence:
  in Haskell, one can define:

```
test :: Maybe a -> Bool
test Nothing = True
test (Just_) = False
```

Product

Definition

$D \times E$ the set of pairs with the orderly relation:
$\langle d_1, e_1 \rangle \sqsubseteq \langle d_2, e_2 \rangle$ iff $d_1 \sqsubseteq d_2$ and $e_1 \sqsubseteq e_2$

It is generalized to the finished product.

Builds CPOs associated with pairs, records, vectors.

Associated functions:

- projections $\pi_1 : (D \times E) \to D$, $\pi_1(\langle d, e \rangle) = d$,
  $\pi_2 : (D \times E) \to E$
- from each pair of functions $f : C \to D$, $g : C \to E$
  we derive $\langle f, g \rangle : C \to (D \times E)$
  $\langle f, g \rangle(c) = \langle f(c), g(c) \rangle$
  (returns pairs of elements).

These functions define an isomorphism between
$(C \to D) \times (C \to E)$ and $C \to (D \times E)$ given by ...

Exercises

- Show that the definitions are well-given: order $D \times E$ is a CPO, the π_i, $\langle f, g \rangle$ functions are continuous.
- Build $O \times O (= O^2)$, O^3. Which orders are isomorphic?
- Build $(T_\perp)^2$

Properties

Proposition

For any indexes set of elements in D
$\{d_{i,j} | i, j \in Nat, \forall\ i \leq i', j \leq j'\ .\ d_{i,j} \sqsubseteq d_{i',j'}\}$

$$\bigsqcup_i d_{i,i} = \bigsqcup_i \bigsqcup_j d_{i,j} = \bigsqcup_j \bigsqcup_i d_{i,j}$$

Proposition

A function $f : (C \times D) \to E$ is continued iff is continued in each of its arguments.

Proposition not true on real numbers: $f(x, y) = \frac{x \cdot y}{x^2 + y^2}$

Function Space

Definition

$[D \to E]$ the set of continuous functions from D to E
with the point-wise order:
$f \sqsubseteq g$ iff for every $d \in D$, $f(d) \sqsubseteq g(d)$.

- To build CPO for functional languages:
  `type FunInt = Integer -> Integer`
- Notice the square brackets [ ] in the notation.
- Prove $[D \to E]$ is a CPO.

Associated functions

- application $app : ([D \to E] \times D) \to E$
  $app(\langle f, d \rangle) = f(d)$
- currying (from Haskell Curry), a function
  $f : (D \times E) \to F$ induces a function
  $curry(f) : D \to [E \to F]$
  $curry(f)(d)(e) = f(\langle d, e \rangle)$

These functions define an isomorphism between $C \to [D \to E]$ and $(C \times D) \to E)$, given by . . .

Properties and exercises

- Show that $[T \to D]$ is isomorphic to D^2.
- Draw CPOs:
  - $[O \to T]$,
  - $[O \to T_\perp]$,
  - $[T_\perp \to T_\perp]$,
- Show that fixed-point operator $Y : [D \to D] \to D$ is continuous.

Disjoint Sum

Definition

$D + E \ = \ \{\langle 1, d\rangle \mid d \in D\} \cup \{\langle 2, e\rangle \mid e \in E\}$
with order:

- $\langle 1, d\rangle \sqsubseteq \langle 1, d'\rangle$ iff $d \sqsubseteq d'$
- $\langle 2, e\rangle \sqsubseteq \langle 2, e'\rangle$ iff $e \sqsubseteq e'$
- $\langle 1, d\rangle$ incomparable with $\langle 2, e\rangle$

Show that $D + E$ is a CPO (without $\perp$).

CPO associated with variant types.

```
data Bool = True | False
```

Associated functions

- insertions: $in_1 : D \to (D + E)$, $in_1(d) = \langle 1, d\rangle$, $in_2 : E \to (D + E)$
- from functions $f : D \to C$ and $g : E \to C$ construct the function $[f, g] : (D + E) \to C$:
  $[f, g](\langle 1, d\rangle) = f(d)$,
  $[f, g](\langle 2, e\rangle) = g(e)$,

A induced isomorphism between $(D \to C) \times (E \to C)$ and $[D + E] \to C$

Exercises

- Define, through standard functions and constructors, the function $cond : (T \times E \times E) \rightarrow E$
  $cond(b, e_1, e_2) = \ldots$
  and the function $(\_ \mid \_, \_) : (T_\perp \times E \times E) \rightarrow E$
  $(b \mid e_1, e_2) = \ldots$
- Define, by means of standard functions and constructors, the semantics of Boolean functions.
- Define the n-ary sum (of n CPO).
  Can one reduce the n-ary to repeated application of binary sum?

Metalanguage

CPO functions, of the form$(D_1 \times \ldots \times D_i) \rightarrow E$ can be constructed from a language that uses:

- variables with a domain type: $x_1 : D_1$, ..., $x_i : D_i$
- Constant: *true*, *false*, $-1, 0, 1, \ldots$
- basic functions: $\lfloor \_ \rfloor$, π_i, *app*, *in*$_i$, *fix*
- builders: $(\_)^*$, $\langle \_, \_ \rangle$, *curry*$(\_)$, $[\_, \_]$,
- application, lambda abstraction, composition of functions, *let* $\_ \Leftarrow \_ \ . \ \_$

Examples:

- $\langle \pi_2, \pi_1 \rangle$
- $\lambda x \ . \ f(g(x)) \quad \lambda x \ . \ f(\pi_1(x))(\pi_2(x)) \quad \lambda(x, y) \ . \ f(x)(y)$

Metalanguage

Proposition

Each expression e, having type E, with variables $x_1 : D_1$, ..., $x_i : D_i$, denotes a continuous function f in each of its variables, that is, $f : (D_1 \times \ldots \times D_i) \to E$ is continuous.

Proof.

By induction on the structure of the e, we define the meaning of e and from here the continuity. $\square$

Metalanguage allows you to define mathematical objects, with a syntax similar to Haskell.

Metalanguage and Haskell define objects of different nature.

Functional Languages

Semantic (operational and denotational) semantics of two simple functional languages

with two different assessment mechanisms

- call-by-value (eager) like: Scheme, Standard ML, OCaml.
- call-by-name (lazy) like Haskell, Miranda.

Expressions

$$
\begin{array}{rl}
t ::= & x \\
& \mathbf{n} \mid t_1 \,\mathbf{op}\, t_2 \mid \\
& (t_1, t_2) \mid \mathbf{fst}\, t \mid \mathbf{snd}\, t \mid \\
& \lambda x.t \mid (t_1\, t_2) \mid \\
& \mathbf{if}\, t_0 \,\mathbf{then}\, t_1 \,\mathbf{else}\, t_2 \mid \\
& \mathbf{let}\, x \Leftarrow t_1 \,\mathbf{in}\, t_2 \mid \\
& \mathbf{rec}\, f.\lambda x.t
\end{array}
$$

Type checking

Not all expressions are meaningful: $(1\ \ 3)$, $((\lambda x.x+1)(2,3))$

Type checking:

- determines the correct expressions,
- derives $t : \tau$
- types:

$$\tau ::= \textbf{int} \mid \tau_1 * \tau_2 \mid \tau_1 \to \tau_2$$

- induction rules on the structure of terms
  - each variable has an associated type x^τ, written simply as x
  - every syntactic construct has an associated rule:

$$\frac{x : \tau_1 \quad t_1 : \tau_1 \quad t_2 : \tau_2}{\textbf{let}\ x \Leftarrow t_1\ \textbf{in}\ t_2 : \tau_2}$$

No polymorphism

- Each variable has associated a single type.
- Property: each expression has a unique type, no polymorphism;
- without polymorphism: type system and denotational semantics are simpler.

Operational Semantics

defines a system of rules describing how a term reduces.

Two alternatives:

- big-step reduction: $t \Rightarrow c$ ($t \rightarrow c$) describes the value c generated by the computation of t,
  closer to denotational semantics.
- small-step reduction $t_1 \rightarrow t_2$ describes how computation steps turn t_1 into t_2.

  In the small step semantic for functional languages, a single step of computation can substitutes several occurrences of a variable by a duplicated term,
  This is not an elementary computing step.
- Small step reduction can be useful for debugging, see Haskell.

Call-by-value, big-step reduction

For each type, a set of values or canonical forms:
computational results,
terms that cannot be further reduced.

- **int** (ground types), the numeric constants $\dots -1, 0, 1, 2 \dots$
- $\tau_1 * \tau_2$, pairs (v_1, v_2), with v_i value.
  Eager reduction, fully defined elements.
- $\tau_1 \to \tau_2$, λ-abstraction $\lambda x.t$, with t not needed a value.
  Alternative definitions are possible: $\lambda x.v$ with v not reduceable term, examples $x + 3$, $(x\ 1)$

By definition values are closed terms: one can only evaluate closed terms; this restriction simplifies the definitions.

Reduction rules: for induction on term structure

$$c \;\Rightarrow\; c \qquad c \text{ canonical form}$$

$$\frac{t_0 \;\Rightarrow\; m \quad t_1 \;\Rightarrow\; n}{t_0 + t_1 \;\Rightarrow\; o} \qquad o = m + n$$

...

Reduction rules: for induction on term structure

$$c \;\Rightarrow\; c \qquad c \text{ canonical form}$$

$$\frac{t_0 \;\Rightarrow\; m \quad t_1 \;\Rightarrow\; n}{t_0 + t_1 \;\Rightarrow\; o} \qquad o = m + n$$

. . .

$$\frac{t_0 \;\Rightarrow\; 0 \quad t_1 \;\Rightarrow\; c}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \;\Rightarrow\; c}$$

Reduction rules: for induction on term structure

$$c \;\Rightarrow\; c \qquad c \text{ canonical form}$$

$$\frac{t_0 \;\Rightarrow\; m \quad t_1 \;\Rightarrow\; n}{t_0 + t_1 \;\Rightarrow\; o} \qquad o = m + n$$

...

$$\frac{t_0 \;\Rightarrow\; 0 \quad t_1 \;\Rightarrow\; c}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \;\Rightarrow\; c}$$

$$\frac{t_0 \;\Rightarrow\; n \qquad t_2 \;\Rightarrow\; c}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \;\Rightarrow\; c} \quad n \neq 0$$

Reduction rules: for induction on term structure

$$c \Rightarrow c \qquad c \text{ canonical form}$$

$$\frac{t_0 \Rightarrow m \quad t_1 \Rightarrow n}{t_0 + t_1 \Rightarrow o} \qquad o = m + n$$

...

$$\frac{t_0 \Rightarrow 0 \quad t_1 \Rightarrow c}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \Rightarrow c}$$

$$\frac{t_0 \Rightarrow n \qquad t_2 \Rightarrow c}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \Rightarrow c} \quad n \neq 0$$

$$\frac{t_1 \Rightarrow c_1 \qquad t_2 \Rightarrow c_2}{(t_1, t_2) \Rightarrow (c_1, c_2)}$$

Reduction rules: for induction on term structure

$$c \;\Rightarrow\; c \qquad c \text{ canonical form}$$

$$\frac{t_0 \;\Rightarrow\; m \quad t_1 \;\Rightarrow\; n}{t_0 + t_1 \;\Rightarrow\; o} \qquad o = m + n$$

...

$$\frac{t_0 \;\Rightarrow\; 0 \quad t_1 \;\Rightarrow\; c}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \;\Rightarrow\; c}$$

$$\frac{t_0 \;\Rightarrow\; n \qquad t_2 \;\Rightarrow\; c}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \;\Rightarrow\; c} \quad n \neq 0$$

$$\frac{t_1 \;\Rightarrow\; c_1 \qquad t_2 \;\Rightarrow\; c_2}{(t_1, t_2) \;\Rightarrow\; (c_1, c_2)}$$

$$\frac{t \;\Rightarrow\; (c_1, c_2)}{\textbf{fst}\, t \;\Rightarrow\; c_1}$$

Reduction rules

$$\frac{t_1 \;\Rightarrow\; \lambda x.t_1' \quad t_2 \;\Rightarrow\; c_2 \quad t_1'[c_2/x] \;\Rightarrow\; c}{(t_1\, t_2) \;\Rightarrow\; c}$$

Reduction rules

$$\frac{t_1 \;\Rightarrow\; \lambda x.t_1' \quad t_2 \;\Rightarrow\; c_2 \quad t_1'[c_2/x] \;\Rightarrow\; c}{(t_1\,t_2) \;\Rightarrow\; c}$$

$$\frac{t_1 \;\Rightarrow\; c_1 \quad t_2[c_1/x] \;\Rightarrow\; c}{\mathbf{let}\,x \Leftarrow t_1\,\mathbf{in}\,t_2 \;\Rightarrow\; c}$$

Reduction rules

$$\frac{t_1 \;\Rightarrow\; \lambda x.t_1' \quad t_2 \;\Rightarrow\; c_2 \quad t_1'[c_2/x] \;\Rightarrow\; c}{(t_1\, t_2) \;\Rightarrow\; c}$$

$$\frac{t_1 \;\Rightarrow\; c_1 \quad t_2[c_1/x] \;\Rightarrow\; c}{\mathbf{let}\, x \Leftarrow t_1 \,\mathbf{in}\, t_2 \;\Rightarrow\; c}$$

$$\mathbf{rec}\, x.\lambda y.t \;\Rightarrow\; \lambda y.t[\mathbf{rec}\, x.\lambda y.t \,/\, x]$$

Reduction property

- (Strong) Deterministic reduction, each term there is a single possible derivation: the term reduces at most to one value. Proof by cases on the form of the term and by inductions on the derivation.
- Reduction preserves types: subject reduction.
- **rec** Endless derivations correspond to infinite computations: example: $((\mathbf{rec}\, f^{\mathbf{N} \to \mathbf{N}}.\, \lambda y.\, (f \quad 1))\ 2)$
- Exercise
  $((\mathbf{rec}\, f.\lambda x.\, \mathbf{if}\ x\ \mathbf{then}\ 0\ \mathbf{else}\ (f(x-1)) + 2)\ 2)$

Domains for Denotational Semantics

separated in:

- domains for interpreting values;
- domains to interpret computations.

Domains for values: by induction on the structure of the type.

- $V_{\mathbf{int}} = N$
- $V_{\tau_1 * \tau_2} = V_{\tau_1} \times V_{\tau_2}$
- $V_{\tau_1 \to \tau_2} = [V_{\tau_1} \to (V_{\tau_2})_\perp]$

Domain for computation:

$$(V_\tau)_\perp$$

Environment

The interpretation of an open term depends on how we interpret variables (from the environment).
In call-by-value languages, variables denote values.
Env, the set of functions from variables in the domains for computations

$$\rho : \mathbf{Var} \to \sum_{\tau \in \mathcal{T}} V_\tau$$

that respect the types $x : \tau$ implies $\rho(x) \in V_\tau$.

The interpretation of a term $t : \tau$,

$$[\![t]\!] : \mathbf{Env} \to (V_\tau)_\perp$$

Inductive definition on terms

$$
\begin{aligned}
[\![x]\!]\rho &= \lfloor \rho(x) \rfloor \\
[\![\mathbf{n}]\!]\rho &= \lfloor n \rfloor
\end{aligned}
$$

Inductive definition on terms

$$[\![x]\!]\rho = \lfloor \rho(x) \rfloor$$

$$[\![\mathbf{n}]\!]\rho = \lfloor n \rfloor$$

$$[\![t_1 \;\mathbf{op}\; t_2]\!]\rho = [\![t_1]\!]\rho \; op_\perp \; [\![t_2]\!]\rho$$

$$[\![\mathbf{if}\; t_0 \;\mathbf{then}\; t_1 \;\mathbf{else}\; t_2]\!]\rho = Cond([\![t_0]\!]\rho, [\![t_1]\!]\rho, [\![t_2]\!]\rho)$$

Inductive definition on terms

$$[\![x]\!]\rho = \lfloor \rho(x) \rfloor$$
$$[\![\mathbf{n}]\!]\rho = \lfloor n \rfloor$$

$$[\![t_1\ \mathbf{op}\ t_2]\!]\rho = [\![t_1]\!]\rho\ op_\perp\ [\![t_2]\!]\rho$$
$$[\![\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2]\!]\rho = Cond([\![t_0]\!]\rho, [\![t_1]\!]\rho, [\![t_2]\!]\rho)$$

$$[\![(t_1, t_2)]\!]\rho = let\ v_1 \Leftarrow [\![t_1]\!]\rho.\ let\ v_2 \Leftarrow [\![t_2]\!]\rho. \lfloor (v_1, v_2) \rfloor$$
$$[\![(\mathbf{fst}\ t)]\!]\rho = let\ v \Leftarrow [\![t]\!]\rho.\ \lfloor \pi_1(v) \rfloor$$

Inductive definition on terms

$$[\![x]\!]\rho = \lfloor \rho(x) \rfloor$$
$$[\![\mathbf{n}]\!]\rho = \lfloor n \rfloor$$

$$[\![t_1 \;\mathbf{op}\; t_2]\!]\rho = [\![t_1]\!]\rho \; op_\perp \; [\![t_2]\!]\rho$$
$$[\![\mathbf{if}\; t_0 \;\mathbf{then}\; t_1 \;\mathbf{else}\; t_2]\!]\rho = Cond([\![t_0]\!]\rho, [\![t_1]\!]\rho, [\![t_2]\!]\rho)$$

$$[\![(t_1, t_2)]\!]\rho = let\; v_1 \Leftarrow [\![t_1]\!]\rho.\, let\; v_2 \Leftarrow [\![t_2]\!]\rho. \lfloor (v_1, v_2) \rfloor$$
$$[\![(\mathbf{fst}\; t)]\!]\rho = let\; v \Leftarrow [\![t]\!]\rho.\, \lfloor \pi_1(v) \rfloor$$

$$[\![\lambda x.t]\!]\rho = \lfloor \lambda v : V_\sigma .\, [\![t]\!](\rho[v/x]) \rfloor$$
$$[\![(t_1\, t_2)]\!]\rho = let\; v_1 \Leftarrow [\![t_1]\!]\rho.\, let\; v_2 \Leftarrow [\![t_2]\!]\rho.\, v_1(v_2)$$

Inductive definition on terms

$$[\![x]\!]\rho = \lfloor \rho(x) \rfloor$$
$$[\![\mathbf{n}]\!]\rho = \lfloor n \rfloor$$

$$[\![t_1 \,\mathbf{op}\, t_2]\!]\rho = [\![t_1]\!]\rho \; op_\perp \; [\![t_2]\!]\rho$$
$$[\![\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2]\!]\rho = Cond([\![t_0]\!]\rho, [\![t_1]\!]\rho, [\![t_2]\!]\rho)$$

$$[\![(t_1, t_2)]\!]\rho = let\ v_1 \Leftarrow [\![t_1]\!]\rho.\, let\ v_2 \Leftarrow [\![t_2]\!]\rho. \lfloor (v_1, v_2) \rfloor$$
$$[\![(\mathbf{fst}\ t)]\!]\rho = let\ v \Leftarrow [\![t]\!]\rho.\, \lfloor \pi_1(v) \rfloor$$

$$[\![\lambda x.t]\!]\rho = \lfloor \lambda v : V_\sigma .\, [\![t]\!](\rho[v/x]) \rfloor$$
$$[\![(t_1\, t_2)]\!]\rho = let\ v_1 \Leftarrow [\![t_1]\!]\rho.\, let\ v_2 \Leftarrow [\![t_2]\!]\rho.\, v_1(v_2)$$

$$[\![\mathbf{rec}\, f.\lambda x.\, t]\!]\rho = \lfloor Fix(\lambda v_1 : V_{\sigma\to\tau}.\, \lambda v_2 : V_\sigma .\, [\![t]\!]\, \rho[v_1/f, v_2/x]) \rfloor$$

Categorical inductive definition on terms

$$[\![x]\!] = \lfloor \_ \rfloor \circ \pi_x$$

$$[\![\mathbf{n}]\!] = \lfloor \_ \rfloor \circ n \circ !$$

$$[\![t_1\,\mathbf{op}\,t_2]\!] \;=\; op_\perp \circ \langle [\![t_1]\!], [\![t_2]\!] \rangle$$

$$[\![(t_1, t_2)]\!] = \lfloor \_ \rfloor \circ \langle [\![t_1]\!], [\![t_2]\!] \rangle$$

$$[\![(\mathbf{fst}\; t)]\!] = [\![\mathbf{fst}]\!] = \lfloor (\lfloor \_ \rfloor \circ \pi_1) \rfloor$$

$$[\![\lambda x.t]\!] = \lfloor \_ \rfloor \circ curry([\![t]\!] \circ \prod_{y \in \mathbf{Var}} (in_y \circ f_y))$$

$$[\![(t_1\,t_2)]\!] = app \;\circ \langle (id)^* \circ [\![t_1]\!], [\![t_2]\!] \rangle$$

$$[\![\mathbf{rec}\, x.\lambda y.\, t]\!] = Fix \circ curry([\![t]\!] \circ \prod_{y \in \mathbf{Var}} (in_y \circ f_y))$$

where $f_x = \pi_1$ and $f_y = \pi_y \circ \pi_2$ if $x \neq y$

Properties of the denotational semantics

Substitution lemma

For each closed term s, if $[\![s]\!]\rho = \lfloor v \rfloor$ then $[\![t[s/x]]\!]\rho = [\![t]\!]\rho[v/x]$,

- Proved by induction on the structure of the term t.
- Instrumental to prove other properties.

Denotational semantics of values

For each value c, $[\![c]\!]_\rho \neq \bot$.

Comparison between semantics: Correctness

Correctness of operational semantics:

Proposition

For each term t, and c value,

$$t \;\Rightarrow\; c \text{ implies } [\![t]\!] = [\![c]\!]$$

- It is proved by induction on the derivation of $t \;\Rightarrow\; c$.
- The rules of operational semantics respect the denotation.

Comparison between semantics: Adequacy

The opposite implication:

$$[\![t]\!] = [\![c]\!] \text{ implies } t \Rightarrow c$$

does not hold, since:
there are two different values c_1 and c_2 with the same denotational semantics.

Therefore: $[\![c_1]\!] = [\![c_2]\!]$ but $c \not\Rightarrow c'$

Example: $\lambda x.0$ and $\lambda x.0 + 0$.

Adequacy

A weaker property holds:

Proposition

For each term t, and value c ,

$$[\![t]\!] = [\![c]\!] \text{ implies the existence of } c' \text{ such that } t \;\Rightarrow\; c'.$$

In other terms: if $[\![t]\!] \neq \bot$, then t terminates, according to the operational semantics.

The proof is not obvious, uses sophisticated techniques: logical relations.

Corollary

For each $t : int$ and integer n

$$t \;\Rightarrow\; n \;\text{ iff }\; [\![t]\!] = [\![n]\!]$$

Call-by-name language

Different recursive definition

$$\mathbf{rec}\, x.t$$

Different sets of values for pair types.

int	the numeric constants $\ldots -1, 0, 1, 2 \ldots$
$\tau_1 * \tau_2$	(t_1, t_2), with t_i closed, not necessarily values. Remark: defining values, on a type list, in this way, one can handle infinite lists. Regardless call-by-value, call-by-name reduction mechanism.
$\tau_1 \rightarrow \tau_2$	closed terms $\lambda x.t$, with t not necessarily a value.

Reduction rules

Differences from call-by-name semantics:

$$\frac{}{(t_1, t_2) \;\Rightarrow\; (t_1, t_2)}$$

$$\frac{t \;\Rightarrow\; (t_1, t_2) \qquad t_1 \;\Rightarrow\; c_1}{\textbf{fst}\, t \;\Rightarrow\; c_1}$$

Reduction rules

Differences from call-by-name semantics:

$$\frac{}{(t_1, t_2) \;\Rightarrow\; (t_1, t_2)}$$

$$\frac{t \;\Rightarrow\; (t_1, t_2) \quad t_1 \;\Rightarrow\; c_1}{\mathbf{fst}\, t \;\Rightarrow\; c_1}$$

$$\frac{t_1 \;\Rightarrow\; \lambda x.t_1' \quad t_1'[t_2/x] \;\Rightarrow\; c}{(t_1\, t_2) \;\Rightarrow\; c}$$

Reduction rules

Differences from call-by-name semantics:

$$\frac{}{(t_1, t_2) \;\Rightarrow\; (t_1, t_2)}$$

$$\frac{t \;\Rightarrow\; (t_1, t_2) \qquad t_1 \;\Rightarrow\; c_1}{\textbf{fst}\, t \;\Rightarrow\; c_1}$$

$$\frac{t_1 \;\Rightarrow\; \lambda x.t_1' \qquad t_1'[t_2/x] \;\Rightarrow\; c}{(t_1\, t_2) \;\Rightarrow\; c}$$

$$\frac{t_2[t_1/x] \;\Rightarrow\; c}{\textbf{let}\, x \Leftarrow t_1\, \textbf{in}\, t_2 \;\Rightarrow\; c}$$

Reduction rules

Differences from call-by-name semantics:

$$\frac{}{(t_1, t_2) \;\Rightarrow\; (t_1, t_2)}$$

$$\frac{t \;\Rightarrow\; (t_1, t_2) \quad t_1 \;\Rightarrow\; c_1}{\mathbf{fst}\, t \;\Rightarrow\; c_1}$$

$$\frac{t_1 \;\Rightarrow\; \lambda x.t_1' \quad t_1'[t_2/x] \;\Rightarrow\; c}{(t_1\, t_2) \;\Rightarrow\; c}$$

$$\frac{t_2[t_1/x] \;\Rightarrow\; c}{\mathbf{let}\, x \Leftarrow t_1 \,\mathbf{in}\, t_2 \;\Rightarrow\; c}$$

$$\frac{t[\mathbf{rec}\, x.t \,/\, x] \;\Rightarrow\; c}{\mathbf{rec}\, x.t \;\Rightarrow\; c}$$

Main differences

- evaluation of call-by-name arguments,
- for uniformity, call-by-name evaluation also for constructor **let**,
- recursion applicable to all elements,
- the different set of values for the couple type force different rules.

Properties preserved:

- deterministic reduction, each term reduces to a value at most, there is more than one applicable rule;
- reductions preserve the type: (subject reduction).

Domains for Denotational Semantics

Domains for values:

- $V_{\mathbf{int}} = N$
- $V_{\tau_1 * \tau_2} = (V_{\tau_1})_\perp \times (V_{\tau_2})_\perp$
- $V_{\tau_1 \to \tau_2} = [(V_{\tau_1})_\perp \to (V_{\tau_2})_\perp]$

Domains for computation.

$$(V_\tau)_\perp$$

Environment. In call-by-name languages, variables denote computations.
Env, the set of functions from variables in the domains for computations

$$\rho : \mathbf{Var} \to \sum_{\tau \in \mathsf{type}} (V_\tau)_\perp$$

that respect the types.

$$[\![t]\!] : \mathbf{Env} \to (V_\tau)_\perp$$

Inductive definition on terms

$$\llbracket x \rrbracket \rho \; = \; \rho(x)$$

$$\llbracket \mathbf{n} \rrbracket \rho \; = \; \lfloor n \rfloor$$

$$\llbracket t_1 \, \mathbf{op} \, t_2 \rrbracket \rho \; = \; \llbracket t_1 \rrbracket \rho \; op_\perp \; \llbracket t_2 \rrbracket \rho$$

$$\llbracket \mathbf{if} \; t_0 \; \mathbf{then} \; t_1 \; \mathbf{else} \; t_2 \rrbracket \rho = \mathit{Cond}(\llbracket t_0 \rrbracket \rho, \llbracket t_1 \rrbracket \rho, \llbracket t_2 \rrbracket \rho)$$

$$\llbracket (t_1, t_2) \rrbracket \rho \; = \; \lfloor (\llbracket t_1 \rrbracket \rho, \llbracket t_2 \rrbracket \rho) \rfloor$$

$$\llbracket (\mathbf{fst} \; t) \rrbracket \rho \; = \; \mathit{let} \; v \Leftarrow \llbracket t \rrbracket \rho. \, \pi_1(v)$$

$$\llbracket \lambda x.t \rrbracket \rho \; = \; \lfloor \lambda v : (V_\sigma)_\perp . \, \llbracket t \rrbracket (\rho[v/x]) \rfloor$$

$$\llbracket (t_1 \, t_2) \rrbracket \rho \; = \; \mathit{let} \; v \Leftarrow \llbracket t_1 \rrbracket \rho. \, v(\llbracket t_2 \rrbracket \rho)$$

$$\llbracket \mathbf{rec} \, x.t \rrbracket \rho \; = \; \mathit{Fix}(\lambda v : (V_\sigma)_\perp \llbracket t \rrbracket (\rho[v/x]))$$

Categorical definition

$$
\begin{aligned}
[\![x]\!] &= \pi_x \\
[\![\mathbf{n}]\!] &= \lfloor \_ \rfloor \circ n \circ 1 \\
[\![t_1\, \mathbf{op}\, t_2]\!] &= op_\perp \circ \langle [\![t_1]\!], [\![t_2]\!] \rangle \\
[\![(t_1, t_2)]\!] &= \lfloor \_ \rfloor \circ \langle [\![t_1]\!], [\![t_2]\!] \rangle \\
[\![(\mathbf{fst}\; t)]\!] &= [\![\mathbf{fst}]\!] \lfloor ((\pi_1)^*) \rfloor \\
[\![\lambda x.t]\!] &= \lfloor \_ \rfloor \circ curry([\![t]\!] \circ \prod_{y \in \mathbf{Var}} (in_y \circ f_y)) \\
[\![(t_1\, t_2)]\!] &= app \circ \langle (id)^* \circ [\![t_1]\!], [\![t_2]\!] \rangle \\
[\![\mathbf{rec}\, x.t]\!] &= Fix \circ curry([\![t]\!] \circ \prod_{y \in \mathbf{Var}} (in_y \circ f_y))
\end{aligned}
$$

where $f_x = \pi_1$ and $f_y = \pi_y \circ \pi_2$ if $x \neq y$

Properties of the denotational semantics

- substitution lemma
  If $[\![s]\!]\rho = v$ then $[\![t[s/x]]\!]\rho = [\![t]\!]\rho[v/x]$
  It is proved by induction on the structure of the term t
- for each value c, $[\![c]\!] \neq \perp$

Comparison of Semantics: Correctness and Adequacy

Proposition (Correctness)

For each term t, and c value:

$$t \;\Rightarrow\; c \text{ implies } [\![t]\!] = [\![c]\!]$$

Proposition (Adequacy)

For each term t, and value c,

$$[\![t]\!] = [\![c]\!] \text{ implies the existence of } c' \text{ such that } t \;\Rightarrow\; c'.$$

Corollary

For each $t : int$ and integer n

$$t \;\Rightarrow\; n \text{ iff } [\![t]\!] = [\![n]\!]$$

Observational equality and full-abstraction

Two terms t_1, t_2 are observationally equivalent (the same for the operational semantics) $t_1 \sim t_2$, if:
for each $C[\ ]$ context

$$C[t_1]\downarrow \;\Leftrightarrow C[t_2]\downarrow$$

from the adequacy theorem:

$$[\![t_1]\!] = [\![t_2]\!] \text{ implies } t_1 \sim t_2$$

Opposite implication (full abstraction) is only true for a sequential denotational semantics.

The existence of a "parallel or" function on CPOs cause full-abstraction to fail.

Semantics usage

- Observationally equivalence is difficult to prove, test on infinite context
- Operational semantics useful to specify language implementation.
- Denotational semantics a first step in reasoning on programs.

Simplified denotational semantics

For the “ call-by-name ” languages,
an observational equivalence using just on integer context $C[\,] : \textbf{int}$

- identifies more terms
- allows a simple denotational semantics
- simpler domains not separating domains of values from domains of computations.

- $D_{\textbf{int}} = (N)_\perp$
- $D_{\tau_1 * \tau_2} = D_{\tau_1} \times D_{\tau_2}$
- $D_{\tau_1 \to \tau_2} = [D_{\tau_1} \to D_{\tau_2}]$

It is important to specify which observations can be done on programs.

Inductive definitions on term

$$
\begin{aligned}
[\![x]\!]\rho &= \rho(x) \\
[\![\mathbf{n}]\!]\rho &= \lfloor n \rfloor \\
[\![t_1 \,\mathbf{op}\, t_2]\!]\rho &= [\![t_1]\!]\rho \; op_\perp \; [\![t_2]\!]\rho \\
[\![(t_1, t_2)]\!]\rho &= ([\![t_1]\!]\rho, [\![t_2]\!]\rho) \\
[\![(\mathbf{fst}\; t)]\!]\rho &= \pi_1([\![t]\!]\rho) \\
[\![\lambda x.t]\!]\rho &= \lambda d : D_\sigma \,.\, [\![t]\!](\rho[d/x]) \\
[\![(t_1\, t_2)]\!]\rho &= [\![t_1]\!]\rho([\![t_2]\!]\rho) \\
[\![\mathbf{rec}\, x.t]\!]\rho &= Fix(\lambda d : D_\sigma \,.\, [\![t]\!](\rho[d/x]))
\end{aligned}
$$

Sum Types

We enrich the functional languages with sum types:

$$\tau ::= \ldots \mid \tau_1 + \tau_2$$

```
data Fig = Circle Integer | Rectangle Integer Integer
```

Constructors and destructors.

$$\mathbf{inl}(t) \mid \mathbf{inr}(t) \mid \mathbf{case}\ t\ \mathbf{of}\ \mathbf{inl}(x_1).t_1,\ \mathbf{inr}(x_2).t_2$$

```
case fig of
 Circle r -> pi * r * r
 Rectangle b h -> b * h
```

Sum Types

- Typing rules: an expression can have more than one type (in contrast with Haskell)
- Operational semantics (eager - lazy)
- Denotational semantic
- Simplified denotational semantics.

Haskell's denotational semantics

We aim to extend the operational and denotational semantics of functional lazy language to a large part of Haskell.
Some critical points.

- Definitions of types (non recursive).
- Recursive Definitions for pattern matching.
- Recursive Definitions of Types.
- Polymorphism

Core (Kernel) Language

When defining the semantics of a language, or its implementation, it is useful to consider a Core Language:

- simple language, reduced set of primitive builders, reduced subset of main language;
- Main language, a super-set of core language, can be (easily) reduce to it;
- Modular approach to implementation, semantics.

From a theoretical point of view, the core languages are highlighted:

- the fundamental mechanisms of computation;
- similarities: differences between languages.

Core Language for Haskell

System FC

- Lambda calculus:
  application and λ-abstraction;
- polymorphism: explicit types, type variable, type quantifiers.
- algebraic (recursive) data types:
  a set of basic functions, constructors and destructors;
- equivalents between types: coercion operators.

Definition of non recursive types

Can be simulated in Fun (our functional language)
we can associate to each type a type in Fun

```
data TypeId  =  Const1  TypeExp11 ... TypeExp1M |
                Const2  TypeExp21 ... TypeExp2N |
                ...
```

Translate to to the Fun type $tr(TypeId)$,

$$\begin{aligned} tr(TypeId) = \; & tr(TExp11) \times \ldots \times tr(TExp1M)) + \\ & tr(TExp21) \times \ldots \times tr(TExp2N)) + \\ & \ldots \end{aligned}$$

Each constructor translate to a function having type:

$$Const1 : tr(TExp11) \rightarrow \ldots \rightarrow tr(TExp1M) \rightarrow tr(TypeId)$$

definable in the Fun.

Denotational semantics for non recursive types

Given the previous translation, the domains for non recursive types are:

$$\begin{aligned} V_{TId} = \; & ((V_{TExp11})_\perp \times \ldots \times (V_{TExp1M})_\perp) + \\ & ((V_{TExp21})_\perp \times \ldots \times (V_{TExp2N})_\perp) + \\ & \ldots \end{aligned}$$

and the semantics of constructors are:

$$[\![\mathtt{Const1}]\!]_\rho = \lfloor \lambda v_1 : (V_{TExp11})_\perp \ldots \lambda v_M : (V_{TExp1M})_\perp . \lfloor in_1 \langle v_1, \ldots, v_M \rangle \rfloor \rfloor$$

Data Types, Constructors and Destructors

```
date Nat = O | S Nat
```

The definition generates two constructor functions having type:

```
O :: Nat
S :: Nat -> Nat
```

a destructor function of type `case`.

Pattern matching

Pattern matching does not belong to the core language:
definitions by pattern matching reduced to the case constructor on a single arguments

Example:

```
add x O = x
add x (S y) = S (add x y)
```

can be translated into the core language by:

```
let add = rec add' . \x y -> case y of O -> x |
                                       S y1 -> S (add' x y1)
in
```

Pattern with more arguments

The definition

```
and True True = True
and _    False = False
and False _    = False
```

can translate into:

```
let and = \x y -> case x of True -> case y of True -> True
                                              False -> Fal
                             False -> False
```

The construct case makes the order of arguments explicit.
Pattern matching allows more elegant, compact definitions.

Pattern matching: another example

The definition :

```
leq O     _ = True
leq (S _) O = False
leq (Sx) (S y) = leq x y
```

can translate into:

```
let leq = rec leq'.
  \x y -> case x of O -> True
                    S x1 -> case y of O -> False
                                      S y1 -> leq' x1 y1
in ...
```

More complex to define mutually recursive functions:
recursively defines an array of functions, one element in a product type.

Recursive data types

Such as the lazy naturals:

```
date Nat = O | S Nat
```

It induce a domain equation:
$Nat \;\cong\; 1 + (Nat)_\bot$

It is sufficient to find CPOs satisfying the equation up to isomorphism, i.e, left and right sides of equation can represent different but isomorphic domains.

Domain equations

We consider the problem of solving domain equation, like:

$$Nat = 1 + (Nat)_\perp$$

In simple cases, the solution can be found by mimicking the fixed point solution on CPOs

- start with the empty domain,
- by applying the constructor, build a sequence of domain, each one included in the next one,
  a sequence of approximation
- take the (cpo completion of the) union as solution of the domain equation.

From recursive type to domain equation:

Depending on the laziness of the language, the same recursive type definition can induce different domain equations:

```
data ListBool = Empty | Cons Bool ListBool
```

$$ListBool \ \cong \ 1 + T_\perp \times (ListBool)_\perp$$

the domain of finite and infinite lists.

$$ListBool = 1 + (Bool \times ListBool)$$

the domain of finite lists.

Domain equations

The simple technique to solve domain equations, does not work for equations containing the function space constructors.
Example in Haskell, one can define the recursive type:

```
date FunBool = Fun [FunBool -> Bool]
```

induced domain equation:

$$\mathit{FunBool} \cong (\mathit{FunBool}_\perp \to T_\perp)$$

A second example, the recursive type:

```
date Untyped = Abst [Untyped -> Untyped]
```

induced domain equation:

$$\mathit{Un} \cong (\mathit{Un}_\perp \to \mathit{Un}_\perp)$$

Domain equations

- Difficult problem, left open for thirty years (Dana Scott 1969).
- There are several techniques for solving domain equations,
- each one requires an extension of CPO theory, presented so far.
- We present the one based on Information System.

Information System

An example of Stone duality: mathematical structures can have alternative descriptions: a object is defined by its properties.
Different presentation of a CPO and its elements.
An element of a CPO is described as a subset of elementary basic information.

An information system $\mathcal{A}$ is given by:

- a set of tokens A (elementary information),
- $Con \subseteq \wp_f(A)$, a consistency predicate,
  Con defines the finite sets of consistent tokens.
- an entailment relation (implication) $\vdash \subseteq (Con \times A)$
  $X \vdash a$ denotes that the information contained in X implies information a

Examples: N, T, 1, $1_\bot$, $T_\bot$, $[T_\bot \to T_\bot]$, $[T_\bot \to [T_\bot \to T_\bot]]$

Information System: definition

An information system $\mathcal{A} = (A, Con, \vdash)$ must meet the following conditions:

1. $Con(X)$ and $Y \subseteq X$ implies $Con(Y)$,
2. for every $a \in A$ we have $Con(\{a\})$,
3. $X \vdash a$ implies $Con(X \cup \{a\})$ and $X \neq \emptyset$
4. $Con(X)$ and $a \in X$ implies $X \vdash a$,
5. $Con(X), Con(Y)$ $X \vdash^* Y$ and $Y \vdash a$ implies $X \vdash a$

   where $X \vdash^* Y$ indicates $\forall b \in Y . X \vdash b$

From Information System to CPO

The CPO $\lceil \mathcal{A} \rceil$ associated with the information system $\mathcal{A} = (A, Con, \vdash)$ is composed by those subset x of A such that:

- x is consistent, i.e. for every $Y \subseteq_f x$, $Con(Y)$,
- x is closed for the entailment relation, i.e.:
  for every $Y \subseteq_f x$, $Y \vdash a$ implies $a \in x$

The order on $\lceil \mathcal{A} \rceil$ is the information order:
$x \sqsubseteq y$ if and only if $x \subseteq y$

$\lceil \mathcal{A} \rceil$ is a CPO with minimum element.

Alternative Definition

An alternative CPO construction $|\mathcal{A}|$ can be obtained by adding, to the previous, the condition

- $x \neq \emptyset$, different from empty.
  - Winskel uses this second definition;
  - $\lceil\mathcal{A}\rceil \cong (|\mathcal{A}|)\bot$;
  - $|\mathcal{A}|$ can be CPO without minimum element $\bot$.
  - $\lceil\mathcal{A}\rceil$ generates the CPO of computations,
    $|\mathcal{A}|$ generates the CPO of values.

Some comments

- Different information systems can induce the same CPO.
- There are CPOs that cannot be defined through an information system.
- The class of CPOs defined by the information system is named Scott's domain (Dana Scott).
  Consistently complete, ω-algebraic CPO.
- There are variants of the information system called coherent spaces:
  - with a binary the coherence relations,
  - no entailment relation.
- Scott's Domain and Information System form an example of Stone Duality,
  elements of a space completely described by their properties.

The empty information system

The empty information system:

$$\mathbf{0} = (\emptyset, \{\emptyset\}, \emptyset)$$

$$\lceil \mathbf{0} \rceil = \{\bot\}$$

Information System Constructors: Lifting

Given the information system: $\mathcal{A} = (A, Con, \vdash)$
define $\mathcal{A}_\perp = (A_\perp, Con_\perp, \vdash_\perp)$
as:

- $A_\perp \; = \; A \uplus \{*\}$
- $Con_\perp(X)$ iff $\exists Con(Y) \, . \, (X = in_l Y \; \vee \; X = (in_l \, Y \cup \{in_r \, *\}))$
- $X \vdash_\perp a$ iff $a = in_r \, * \; \vee$
  $\exists (Y \vdash b) \, . \, a = in_l b \; \wedge \; (X = in_l Y \; \vee \; X = in_l Y \cup \{in_r *\}$

Proposition

$$|\mathcal{A}_\perp| \cong |\mathcal{A}|_\perp \qquad\qquad \lceil \mathcal{A}_\perp \rceil \cong \lceil \mathcal{A} \rceil_\perp$$

Definition: $\mathbf{1} = \mathbf{0}_\perp$

Examples: $\mathbf{T}_\perp$ $\qquad (\mathbf{T}_\perp)_\perp$

Product

Given: $\mathcal{A} = (A, Con_A, \vdash_A)$ e $\mathcal{B} = (B, Con_B, \vdash_B)$
define: $\mathcal{A} \times \mathcal{B} = (A \uplus B, Con_{A\times B}, \vdash_{A\times B})$
as

- $Con_{A\times B}(in_l\, X \cup in_r\, Y)$ iff $Con_A(X)$ e $Con_B(Y)$
- $(in_l\, X \cup in_r\, Y) \vdash_{A\times B} (in_l\, a)$ iff $X \vdash_A a$
  $(in_l\, X \cup in_r\, Y) \vdash_{A\times B} (in_r\, b)$ iff $Y \vdash_B b$

Proposition

$$\lceil \mathcal{A} \times \mathcal{B} \rceil \cong \lceil \mathcal{A} \rceil \times \lceil \mathcal{B} \rceil$$

Examples: $\mathbf{T} \times \mathbf{T}$ $\qquad$ $\mathbf{T}_\perp \times \mathbf{T}_\perp$

Coalescent product

Given: $\mathcal{A} = (A, Con_A, \vdash_A)$ e $\mathcal{B} = (B, Con_B, \vdash_B)$
define: $\mathcal{A} \otimes \mathcal{B} = (A \times B, Con_{A\otimes B}, \vdash_{A\otimes B})$
as:

- $Con_{A\otimes B}(Z)$ iff $Con_A(\pi_1 Z)$ e $Con_B(\pi_2 Z)$
- $Z \vdash_{A\otimes B} c$ iff $\pi_1 Z \vdash_A \pi_1 c \quad \wedge \quad \pi_2 Z \vdash_A \pi_2 c$

Proposition

$$|\mathcal{A} \otimes \mathcal{B}| \cong |\mathcal{A}| \times |\mathcal{B}|$$

Examples: $\mathbf{T} \otimes \mathbf{T} \qquad \mathbf{T}_\perp \otimes \mathbf{T}_\perp$

Sum

Given the IS $\mathcal{A} = (A, Con_A, \vdash_A)$ e $\mathcal{B} = (B, Con_B, \vdash_B)$ define $\mathcal{A} + \mathcal{B} = (A \uplus B, Con_{A+B}, \vdash_{A+B})$ as:

- $Con_{A \times B}(X)$ iff $X = in_l\, Y \;\wedge\; Con_A(Y)$
  or $X = in_r\, Y \;\wedge\; Con_B(Y)$
- $(in_l\, X) \vdash_{A+B} (in_l a)$ iff $X \vdash_A a$
  $(in_r\, Y) \vdash_{A_B} (in_r b)$ iff $Y \vdash_B b$

Proposition

$$\lceil \mathcal{A} + \mathcal{B} \rceil \cong \text{ coalescent sum of } \lceil \mathcal{A} \rceil, \lceil \mathcal{B} \rceil$$

$$|\mathcal{A} + \mathcal{B}| \cong |\mathcal{A}| + |\mathcal{B}|$$

Definion: $\mathbf{T} = \mathbf{2} = \mathbf{1} + \mathbf{1}$

Examples: $\mathbf{T} + \mathbf{T}$ $\qquad$ $\mathbf{T}_\perp + \mathbf{T}_\perp$

Function space

Given $\mathcal{A} = (A, Con_A, \vdash_A)$ e $\mathcal{B} = (B, Con_B, \vdash_B)$
define $\mathcal{A} \to \mathcal{B} = (C, Con_C, \vdash_C)$
as:

- $C \;=\; \{(X, b) \mid X \in Con_A, b \in B\}$
- $Con_C(\{(X_1, b_1), \ldots, (X_n, b_n)\})$ iff $\forall I \subseteq \{1, \ldots, n\}\,.$
  $Con_A(\bigcup_{i \in I} X_i) \Rightarrow Con_B(\{b_i \mid i \in I\})$
- $\{(X_1, b_1), \ldots (X_n, b_n)\} \vdash_C (X, b)$ iff $\{b_i \mid (X \vdash_A^* X_i)\} \vdash_B b$

Proposition

$$\lceil \mathcal{A} \to \mathcal{B} \rceil \cong [\lceil \mathcal{A} \rceil \to \lceil \mathcal{B} \rceil]$$

Examples: $\mathbf{T} \to \mathbf{T}$ $\qquad$ $\mathbf{T}_\perp \to \mathbf{T}_\perp$

Strict function space

The space of strict (continuous) functions $[C \to_\perp D]$:
the functions f st $f(\perp) = \perp$
is described by the IS that repeats the previous construction by modifying only the definition of the first point:
$C \;=\; \{(X, b) \mid X \in Con_A, X \neq \emptyset, b \in B\}$

Proposition

$$\lceil \mathcal{A} \to_\perp \mathcal{B} \rceil \;\cong\; [\lceil \mathcal{A} \rceil \to_\perp \lceil \mathcal{B} \rceil] \;\cong\; [|\mathcal{A}| \to |(\mathcal{B})_\perp|]$$

Examples: $\mathbf{T} \to_\perp \mathbf{T}$ $\qquad\qquad$ $\mathbf{T}_\perp \to_\perp \mathbf{T}_\perp$

Denotational semantics via information systems

IS allows an alternative, dual, finitary, presentation of semantics of terms.

As set of rules deriving judgements in the form:

$$\Gamma \vdash t : a$$

stating, under the assumption Γ the token a belong to the semantics of t.

Γ has form $x : \{a_1, \dots a_n\}, \dots, y : \{b_1, \dots b_n\}$
x, y, possibly, free variable in t
assume that the interpretation of x contains the token $a_1, \dots a_n$.

Rules

$$\frac{\Gamma \vdash t_0 : \overline{m} \quad \Gamma \vdash t_1 : \overline{n}}{\Gamma \vdash t_0 + t_1 : \overline{o}} \qquad o = m + n$$

...

Rules

$$\frac{\Gamma \vdash t_0 : \overline{m} \quad \Gamma \vdash t_1 : \overline{n}}{\Gamma \vdash t_0 + t_1 : \overline{o}} \qquad o = m + n$$

...

$$\frac{\Gamma \vdash t_0 : \overline{0} \quad \Gamma \vdash t_1 : a}{\Gamma \vdash \textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 : a}$$

Rules

$$\frac{\Gamma \vdash t_0 : \overline{m} \quad \Gamma \vdash t_1 : \overline{n}}{\Gamma \vdash t_0 + t_1 : \overline{o}} \qquad o = m + n$$

...

$$\frac{\Gamma \vdash t_0 : \overline{0} \quad \Gamma \vdash t_1 : a}{\Gamma \vdash \textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 : a}$$

$$\frac{\Gamma \vdash t_0 : \overline{n} \qquad \Gamma \vdash t_2 : a}{\Gamma \vdash \textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 : c} \quad n \neq 0$$

Rules

$$\frac{\Gamma \vdash t_0 : \overline{m} \quad \Gamma \vdash t_1 : \overline{n}}{\Gamma \vdash t_0 + t_1 : \overline{o}} \qquad o = m + n$$

...

$$\frac{\Gamma \vdash t_0 : \overline{0} \quad \Gamma \vdash t_1 : a}{\Gamma \vdash \textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 : a}$$

$$\frac{\Gamma \vdash t_0 : \overline{n} \qquad \Gamma \vdash t_2 : a}{\Gamma \vdash \textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 : c} \quad n \neq 0$$

$$\frac{\Gamma \vdash t_1 : a}{\Gamma \vdash (t_1, t_2) : in_l(a)}$$

Rules

$$\frac{\Gamma \vdash t_0 : \overline{m} \quad \Gamma \vdash t_1 : \overline{n}}{\Gamma \vdash t_0 + t_1 : \overline{o}} \qquad o = m + n$$

...

$$\frac{\Gamma \vdash t_0 : \overline{0} \quad \Gamma \vdash t_1 : a}{\Gamma \vdash \textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 : a}$$

$$\frac{\Gamma \vdash t_0 : \overline{n} \qquad \Gamma \vdash t_2 : a}{\Gamma \vdash \textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 : c} \quad n \neq 0$$

$$\frac{\Gamma \vdash t_1 : a}{\Gamma \vdash (t_1, t_2) : in_l(a)}$$

$$\frac{\Gamma \vdash t : in_l(a)}{\Gamma \vdash \textbf{fst}\, t \ : a}$$

Rules

$$\frac{\Gamma, x : \{a_1, \ldots, a_i\} \vdash t : b}{\Gamma \vdash \lambda x.t_1 : (\{a_1, \ldots, a_i\}, b)}$$

Rules

$$\frac{\Gamma, x : \{a_1, \ldots, a_i\} \vdash t : b}{\Gamma \vdash \lambda x.t_1 : (\{a_1, \ldots, a_i\}, b)}$$

$$\frac{\Gamma \vdash t_1 : (\{a_1, \ldots, a_i\}, b) \quad \Gamma \vdash t_2 : a_1 \quad \ldots \quad \Gamma \vdash t_2 : a_i}{\Gamma \vdash (t_1 t_2) \ : b}$$

Rules

$$\frac{\Gamma, x : \{a_1, \ldots, a_i\} \vdash t : b}{\Gamma \vdash \lambda x.t_1 : (\{a_1, \ldots, a_i\}, b)}$$

$$\frac{\Gamma \vdash t_1 : (\{a_1, \ldots, a_i\}, b) \quad \Gamma \vdash t_2 : a_1 \quad \ldots \quad \Gamma \vdash t_2 : a_i}{\Gamma \vdash (t_1 t_2) \ : b}$$

$$\frac{\Gamma, x : \{a_1, \ldots, a_i\} \vdash t_1 : b \quad \Gamma \vdash t_2 : a_1 \quad \ldots \quad \Gamma \vdash t_2 : a_i}{\Gamma \vdash \textbf{let}\, x \Leftarrow t_2 \,\textbf{in}\, t_1 : b}$$

Rules

$$\frac{\Gamma, x : \{a_1, \ldots, a_i\} \vdash t : b}{\Gamma \vdash \lambda x.t_1 : (\{a_1, \ldots, a_i\}, b)}$$

$$\frac{\Gamma \vdash t_1 : (\{a_1, \ldots, a_i\}, b) \quad \Gamma \vdash t_2 : a_1 \quad \ldots \quad \Gamma \vdash t_2 : a_i}{\Gamma \vdash (t_1 t_2) \ : b}$$

$$\frac{\Gamma, x : \{a_1, \ldots, a_i\} \vdash t_1 : b \quad \Gamma \vdash t_2 : a_1 \quad \ldots \quad \Gamma \vdash t_2 : a_i}{\Gamma \vdash \mathbf{let}\, x \Leftarrow t_2\, \mathbf{in}\, t_1 : b}$$

$$\frac{\Gamma, x : \{a_1, \ldots, a_i\} \vdash t : b \quad \Gamma \vdash \mathbf{rec}\, x.t : a_1 \quad \ldots \quad \Gamma \vdash \mathbf{rec}\, x.t : a_i}{\Gamma \vdash \mathbf{rec}\, x.t : b}$$

An Order on IS

We define an order relation, $\trianglelefteq$, on IS as:

$$(A, Con_A, \vdash_A) \trianglelefteq (B, Con_B, \vdash_B)$$

iff

- $A \subseteq B$
- $Con_A(X) \quad \text{iff} \quad X \subseteq A \ \wedge \ Con_B(X)$
- $X \vdash_A a \quad \text{iff} \quad (X \cup \{a\}) \subseteq A \ \wedge \ X \vdash_B a$

Proposition

The IS set with $\trianglelefteq$ forms a CPO with $\perp$,
the minimum element is $\mathcal{O}$,
the limit of a chain $\mathcal{A}_0 \trianglelefteq \mathcal{A}_1 \trianglelefteq \mathcal{A}_2 \ldots$ with $\mathcal{A}_i = (A_i, Con_i, \vdash_i)$
is obtained through union:
$\bigsqcup_{i\in\omega} \mathcal{A}_i = (\bigcup_{i\in\omega} A_i, \ \bigcup_{i\in\omega} C_i, \ \bigcup_{i\in\omega} \vdash_i \ )$

Continuity

Proposition

The constructors of IS $_\perp$, $+$, $\times$, $\otimes$, $\rightarrow$, $\rightarrow_\perp$ are continuous w.r.t. $\trianglelefteq$.

Corollary

Each constructor build using base constructors is continuous and admits fixed-point.

Eager language with recursive types

Types:

$$\mathbf{1} \mid \tau_1 * \tau_2 \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid X \mid \mu X.\tau$$

Expressions

$$\begin{aligned} t ::= \;& x \\ & () \mid (t_1, t_2) \mid \mathbf{fst}\, t \mid \mathbf{snd}\, t \mid \\ & \lambda x.t \mid (t_1\, t_2) \mid \\ & \mathbf{inl}(t) \mid \mathbf{inr}(t) \mid \mathbf{case}\ t\ \mathbf{of}\ \mathbf{inl}(x_1).t_1,\ \mathbf{inr}(x_2).t_2 \mid \\ & \mathbf{abs}\,(t) \mid \mathbf{rep}\,(t) \mid \\ & \mathbf{rec}\, f.\lambda x.t \end{aligned}$$

One basic type **1**. Nat defined a recursive type.

Type checking

- Each variable has associated a single type, $x_\tau \;\; \to x$;
- induction rules on the term of the term,
  each syntactic construct has a rule of type;
- new rules: unit:
  $$() : \mathbf{1}$$
  abstraction:
  $$\frac{t : \tau[\mu X.\tau/X]}{\textbf{abs}\,(t) : \mu X.\tau}$$
  representation:
  $$\frac{t : \mu X.\tau}{\textbf{rep}\,(t) : \tau[\mu X.\tau/X]}$$
- type conversion, explicit casting type;
- necessary to ensure the uniqueness of the type;
- if the CPO associated with $\mu X.\tau$ is isomorphic to that associated with $\tau[\mu X.\tau/X]$,

Operational Semantics: Set of Values

Because of recursive types
must be defined by a set of inductive rules,
and not by induction on the structure of the type

$$\frac{c : C_{\tau[\mu X.\tau/X]}}{\mathbf{abs}\,(c) : C_{\mu X.\tau}}$$

Reduction rules

New rules for terms in recursive types.

$$\frac{t \;\Rightarrow\; c}{\mathbf{abs}\,(t) \;\Rightarrow\; \mathbf{abs}\,(c)}$$

$$\frac{t \;\Rightarrow\; \mathbf{abs}\,(c)}{\mathbf{rep}\,(t) \;\Rightarrow\; c}$$

Denotational Semantics

By information system

Types of associations - information system made by:

- a type environment (type environment);
- a set of definitions, by induction on the type structure;

$$\begin{aligned}
\mathcal{V}[\![\mathbf{1}]\!]_\chi &= \mathbf{0}_\perp \cong (\{*\}, \{\{*\}, \emptyset\}, \{\{*\} \vdash *\}) \\
\mathcal{V}[\![\tau_1 * \tau_2]\!]_\chi &= \mathcal{V}[\![\tau_1]\!]_\chi \otimes \mathcal{V}[\![\tau_2]\!]_\chi \\
\mathcal{V}[\![\tau_1 + \tau_2]\!]_\chi &= \mathcal{V}[\![\tau_1]\!]_\chi + \mathcal{V}[\![\tau_2]\!]_\chi \\
\mathcal{V}[\![\tau_1 \to \tau_2]\!]_\chi &= (\mathcal{V}[\![\tau_1]\!]_\chi \to_\perp \mathcal{V}[\![\tau_2]\!]_\chi)_\perp \\
\mathcal{V}[\![X]\!]_\chi &= \chi(X) \\
\mathcal{V}[\![\mu X.\tau]\!]_\chi &= \mu I.\mathcal{V}[\![\tau]\!]_{\chi[I/X]}
\end{aligned}$$

Note: $\lceil \mathcal{V}[\![\tau]\!] \rceil$ defines the computing CPO associated with τ,
and $|\mathcal{V}[\![\tau]\!]|$ defines the value CPO associated with τ,
To simplify, we do not define two information systems
(computations and values),

Denotational Semantics

- The equations should be reformulated to fit the domains defined in terms of information systems.
  - the elements of $|\mathcal{V}[\![\tau]\!]_\chi|$ are just set of token, no structure
  - the tokens, elements of $\mathcal{V}[\![\tau]\!]_\chi$ contains the structures, t
- **abs** and **rep** are represented by the identity function.
- The usual properties of correctness and adequacy apply.
- Semantics can be defined considering only the tokens, through a set of rules, in Types Assignment System style.

Denotational semantics via Types Assignment System

As set of rules deriving judgements in the form:

$$\Gamma \vdash t : a$$

stating, under the assumption Γ the token a belong to the semantics of t.

Γ has form $x : \{a_1, \dots a_n\}, \dots, y : \{b_1, \dots b_n\}$
stating that the interpretation of x ($\rho(x)$) contains the token $a_1, \dots a_n$
i.e. $\{a_1, \dots a_n\} \subseteq \rho(x)$

Rules

$$\frac{}{\Gamma \vdash () : *}$$

Rules

$$\frac{}{\Gamma \vdash () : *}$$

$$\frac{1 \leq i \leq n}{\Gamma, x : \{a_1, \ldots, a_n\} \vdash x : a_i}$$

Rules

$$\frac{}{\Gamma \vdash () : *}$$

$$\frac{1 \leq i \leq n}{\Gamma, x : \{a_1, \ldots, a_n\} \vdash x : a_i}$$

$$\frac{\Gamma \vdash t_1 : a_1 \qquad \Gamma \vdash t_2 : a_2}{\Gamma \vdash (t_1, t_2) : (a_1, a_2)}$$

Rules

$$\frac{}{\Gamma \vdash () : *}$$

$$\frac{1 \leq i \leq n}{\Gamma, x : \{a_1, \ldots, a_n\} \vdash x : a_i}$$

$$\frac{\Gamma \vdash t_1 : a_1 \qquad \Gamma \vdash t_2 : a_2}{\Gamma \vdash (t_1, t_2) : (a_1, a_2)}$$

$$\frac{\Gamma \vdash t : (a_1, a_2)}{\Gamma \vdash \textbf{fst}\, t \ : a_1}$$

Rules

$$\frac{\Gamma, x : \{a_1, \ldots, a_i\} \vdash t : b}{\Gamma \vdash \lambda x.t : (\{a_1, \ldots, a_i\}, b)}$$

$$\frac{}{\Gamma \vdash \lambda x.t : *}$$

$$\frac{\Gamma \vdash t_1 : (\{a_1, \ldots, a_i\}, b) \quad \Gamma \vdash t_2 : a_1 \quad \ldots \quad \Gamma \vdash t_2 : a_i}{\Gamma \vdash (t_1 t_2) \ : b}$$

Rules

$$\frac{\Gamma, x : \{a_1, \ldots, a_i\} \vdash t : b}{\Gamma \vdash \lambda x.t : (\{a_1, \ldots, a_i\}, b)}$$

$$\frac{}{\Gamma \vdash \lambda x.t : *}$$

$$\frac{\Gamma \vdash t_1 : (\{a_1, \ldots, a_i\}, b) \quad \Gamma \vdash t_2 : a_1 \quad \ldots \quad \Gamma \vdash t_2 : a_i}{\Gamma \vdash (t_1 t_2) \ : b}$$

$$\frac{\Gamma \vdash \lambda x.t : a_1 \quad \Gamma, f : \{a_1\} \vdash \lambda x.t : a_2 \quad \ldots \quad \Gamma, f : \{a_1, \ldots, a_{i-1}\} \vdash \lambda x.t : a}{\Gamma \vdash \mathbf{rec}\, f.\lambda x.t : a_i}$$

Rules

$$\frac{\Gamma \vdash t : a}{\Gamma \vdash \textbf{inl}(t) : \langle 1, a \rangle}$$

....

$$\frac{\Gamma, x_1 : \{a_1, \ldots, a_i\} \vdash t_1 : b \quad \Gamma \vdash t : \langle 1, a_1 \rangle \quad \ldots \quad \Gamma \vdash t : \langle 1, a_i \rangle}{\Gamma \vdash \textbf{case}\ t\ \textbf{of}\ \textbf{inl}(x_1).t_1,\ \textbf{inr}(x_2).t_2\ \ : b}$$

....

Properties

The usual:

- correctness: reduction preserve denotational semantics;
- adequacy: terms whose denotation is different from $\perp$, converge to a value.

Lazy language with recursive types

Types:

$$\mathbf{0} \mid \tau_1 * \tau_2 \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid X \mid \mu X.\tau$$

Expressions

$$\begin{aligned} t ::= \; & x \\ & \bullet \mid (t_1, t_2) \mid \textbf{fst}\, t \mid \textbf{snd}\, t \mid \\ & \lambda x.t \mid (t_1\, t_2) \mid \\ & \textbf{inl}(t) \mid \textbf{inr}(t) \mid \textbf{case}\ t\ \textbf{of}\ \textbf{inl}(x_1).t_1,\ \textbf{inr}(x_2).t_2 \mid \\ & \textbf{abs}\,(t) \mid \textbf{rep}\,(t) \mid \\ & \textbf{rec}\, x.t \end{aligned}$$

$\mathbf{0}$ does not contain any value, $\bullet$ Divergent term, no reduction rule.

Operational Semantics

The eager rule still applies.

$$\frac{c : C_{\tau[\mu X.\tau/X]}}{\mathbf{abs}\,(c) : C_{\mu X.\tau}}$$

While, values are:

$$(t_1, t_2),\ \mathbf{inl}(t),\ \mathbf{inr}(t)$$

Eager rules for terms in recursive types apply:

$$\frac{t \ \Rightarrow\ c}{\mathbf{abs}\,(t)\ \Rightarrow\ \mathbf{abs}\,(c)}$$

$$\frac{t\ \Rightarrow\ \mathbf{abs}\,(c)}{\mathbf{rep}\,(t)\ \Rightarrow\ c}$$

Denotational Semantics

$$\mathcal{V}[\![\mathbf{0}]\!]_\chi = \mathcal{O}$$
$$\mathcal{V}[\![\tau_1 * \tau_2]\!]_\chi = (\mathcal{V}[\![\tau_1]\!]_\chi \times \mathcal{V}[\![\tau_2]\!]_\chi)_\perp$$
$$\mathcal{V}[\![\tau_1 + \tau_2]\!]_\chi = (\mathcal{V}[\![\tau_1]\!]_\chi)_\perp + (\mathcal{V}[\![\tau_2]\!]_\chi)_\perp$$
$$\mathcal{V}[\![\tau_1 \to \tau_2]\!]_\chi = (\mathcal{V}[\![\tau_1]\!]_\chi \to \mathcal{V}[\![\tau_2]\!]_\chi)_\perp$$
$$\mathcal{V}[\![X]\!]_\chi = \chi(X)$$
$$\mathcal{V}[\![\mu X.\tau]\!]_\chi = \mu I.\mathcal{V}[\![\tau]\!]_{\chi[I/X]}$$

Rules

$$\frac{1 \leq i \leq n}{\Gamma, x : \{a_1, \ldots, a_n\} \vdash x : a_i}$$

Rules

$$\frac{1 \leq i \leq n}{\Gamma, x : \{a_1, \ldots, a_n\} \vdash x : a_i}$$

$$\Gamma \vdash (t_1, t_2) : *$$

$$\frac{\Gamma \vdash t_1 : a_1}{\Gamma \vdash (t_1, t_2) : in_l a_1}$$

Rules

$$\frac{1 \leq i \leq n}{\Gamma, x : \{a_1, \ldots, a_n\} \vdash x : a_i}$$

$$\Gamma \vdash (t_1, t_2) : *$$

$$\frac{\Gamma \vdash t_1 : a_1}{\Gamma \vdash (t_1, t_2) : in_l a_1}$$

$$\frac{\Gamma \vdash t : in_l a_1}{\Gamma \vdash \textbf{fst}\, t : a_1}$$

Rules

$$\frac{\Gamma, x : \{a_1, \ldots, a_i\} \vdash t : b}{\Gamma \vdash \lambda x.t : (\{a_1, \ldots, a_i\}, b)}$$

$\{a_1, \ldots, a_i\}$ can be $\emptyset$

$$\frac{}{\Gamma \vdash \lambda x.t : *}$$

Rules

$$\frac{\Gamma, x : \{a_1, \ldots, a_i\} \vdash t : b}{\Gamma \vdash \lambda x.t : (\{a_1, \ldots, a_i\}, b)}$$

$\{a_1, \ldots, a_i\}$ can be $\emptyset$

$$\frac{}{\Gamma \vdash \lambda x.t : *}$$

$$\frac{\Gamma \vdash t_1 : (\{a_1, \ldots, a_i\}, b) \quad \Gamma \vdash t_2 : a_1 \quad \ldots \quad \Gamma \vdash t_2 : a_i}{\Gamma \vdash (t_1 t_2) \ : b}$$

Rules

$$\frac{\Gamma, x : \{a_1, \ldots, a_i\} \vdash t : b}{\Gamma \vdash \lambda x.t : (\{a_1, \ldots, a_i\}, b)}$$

$\{a_1, \ldots, a_i\}$ can be $\emptyset$

$$\frac{}{\Gamma \vdash \lambda x.t : *}$$

$$\frac{\Gamma \vdash t_1 : (\{a_1, \ldots, a_i\}, b) \quad \Gamma \vdash t_2 : a_1 \quad \ldots \quad \Gamma \vdash t_2 : a_i}{\Gamma \vdash (t_1 t_2) \ : b}$$

$$\frac{\Gamma \vdash t : a_1 \quad \Gamma, x : \{a_1\} \vdash t : a_2 \quad \ldots \quad \Gamma, x : \{a_1, \ldots, a_{i-1}\} \vdash t : a_i}{\Gamma \vdash \mathbf{rec}\, x.t : a_i}$$

Concurrent languages

A link to the seconod part of course:

- concurrent languages: Dijkstra guarded command, CSP, CCS.
- reactive system: CCS, π-calculus

Two communication paradigms among concurrent program, processes:

- shared memory: multiprocessors;
- message exchange: multicomputer, distributed systems.

A simple concurrent language

IMP $+$ parallel composition: $\|$

- no determinism: concurrent programs are intrinsically non-deterministic;
- a program cannot be described as an input-output function.

Concurrent languages semantics

- Operational semantics:
    - shared memory: small-step reduction,
    - message exchange: label transition system
- Denotational semantics: powerdomains, programs like trees,

Dijkstra guarded commands: non deterministic choice

$$c := \textbf{skip} \mid \textbf{abort} \mid X := a \mid c_0; c_1 \mid \textbf{if } gc \textbf{ fi} \mid \textbf{do } gc \textbf{ od}$$

Guarded commands: list of programs with guard,

$$gc := b \rightarrow c \mid gc [] gc$$

- Where b is a Boolean expression.
- Executing a list of guarded commands involves selecting a guard and executing its command.
- Execution fails if no guard is true.

Examples: maximum, Euclid's algorithm.

Small step operational semantics

$$\langle \mathbf{skip}, \sigma\rangle \;\rightarrow\; \sigma$$

$$\frac{\langle a, \sigma\rangle \;\rightarrow\; n}{\langle X := a,\ \sigma\rangle \;\rightarrow\; \sigma[n/x]}$$

$$\frac{\langle c_0, \sigma\rangle \;\rightarrow\; \sigma'}{\langle c_0; c_1,\ \sigma\rangle \;\rightarrow\; \langle c_1,\ \sigma'\rangle}$$

$$\frac{\langle c_0, \sigma\rangle \;\rightarrow\; \langle c_0', \sigma'\rangle}{\langle c_0; c_1,\ \sigma\rangle \;\rightarrow\; \langle c_0'; c_1,\ \sigma'\rangle}$$

$$\frac{\langle gc, \sigma\rangle \;\rightarrow\; \langle c, \sigma'\rangle}{\langle \mathbf{if}\ gc\ \mathbf{fi}, \sigma\rangle \;\rightarrow\; \langle c, \sigma'\rangle}$$

$$\frac{\langle gc, \sigma\rangle \;\rightarrow\; \langle c, \sigma'\rangle}{\langle \mathbf{do}\ gc\ \mathbf{od}, \sigma\rangle \;\rightarrow\; \langle c; \mathbf{do}\ gc\ \mathbf{od},\ \sigma'\rangle}$$

$$\frac{\langle gc, \sigma\rangle \;\rightarrow\; \mathbf{fail}}{\langle \mathbf{do}\ gc\ \mathbf{od}, \sigma\rangle \;\rightarrow\; \sigma}$$

Rules for guarded commands

$$\frac{\langle b,\ \sigma\rangle \ \rightarrow\ \mathbf{true}}{\langle b \rightarrow c,\ \sigma\rangle \ \rightarrow\ \langle c,\ \sigma\rangle}$$

$$\frac{\langle b,\ \sigma\rangle \ \rightarrow\ \mathbf{false}}{\langle b \rightarrow c,\ \sigma\rangle \ \rightarrow\ \mathbf{fail}}$$

$$\frac{\langle gc_0,\ \sigma\rangle \ \rightarrow\ \langle c,\ \sigma\rangle}{\langle gc_0 \mathbin{[\!]} gc_1,\ \sigma\rangle \ \rightarrow\ \langle c,\ \sigma\rangle}$$

$$\frac{\langle gc_1,\ \sigma\rangle \ \rightarrow\ \langle c,\ \sigma\rangle}{\langle gc_0 \mathbin{[\!]} gc_1,\ \sigma\rangle \ \rightarrow\ \langle c,\ \sigma\rangle}$$

$$\frac{\langle gc_0,\ \sigma\rangle \ \rightarrow\ \mathbf{fail} \qquad \langle gc_1,\ \sigma\rangle \ \rightarrow\ \mathbf{fail}}{\langle gc_0 \mathbin{[\!]} gc_1,\ \sigma\rangle \ \rightarrow\ \mathbf{fail}}$$

Dijkstra Guarded commands

- Operational semantics:
    - Small step
    - Big step is possible, the guarded commands do not interact one with the other
- Non determinism allows a cleaner definition of some algorithm, where multiple choices arise naturally,
  see previous examples.

Hoare, CSP Communicating Sequential Processes

Dijkstra guarded command with:

- parallelism $\|$: $c : c_1 \| c_2 \ | \ \ldots$
  but parallel commands communicate through channels, they cannot share variables.
- Communications (synchronous) on channels:
  $$c := \alpha!a \ | \ \alpha?X \ | \ \ldots$$
- Channel restriction:
  $$c := c_1 \backslash \alpha \ | \ \ldots$$
  some channels can only be used within the command.
- Guarded commands with communication: its command executed only if communication
  $$gc \ ::= \ b \wedge \alpha!a \rightarrow c \ | \ b \wedge \alpha?X \rightarrow c \ | \ \ldots$$

In the original CSP, communications are made by specifying the name of a process, not a channel.

Operational semantics of CSP

Label transition system, two types of transactions:

- $\rightarrow$ ($\xrightarrow{\tau}$) describes a computational step,
- $\overset{\alpha!n}{\rightarrow}$ $\overset{\alpha?n}{\rightarrow}$, describes a potential interaction with the exterior.

Reasons for the second type of transaction:

- More synthetic presentation of the transition system.
- Compositional semantics. One describes the single process.

Operational Semantics Rules:

Example: define a process simulating a buffer (with capacity 2).

CSP ideas can be found in GO and Clojure.

Robin Milner, (value-passing) CCS Calculation of Concurrent Systems

A restricted version of CSP. The essential constructor to study concurrency.
Eliminate assignment, narrow the test and cycle commands.

$p :=$	$0 \mid$	empty process
	$b \to p \mid$	boolean guard
	$\alpha!a \to p \mid \alpha?x \to p \mid$	output - input
	$\tau \to p \mid$	empty action
	$p_0 + p_1 \mid$	choice
	$p_0 \parallel p_1 \mid$	parallelism
	$p\backslash\alpha \mid$	restriction
	$p[f] \mid$	relabelling
	$P(a_1, \ldots, a_n)$	process identifiers

A language for describing concurrent systems.

Relation with CSP, what is lost, what can be recovered

skip becomes 0
abort non correspondence
$X := a$ through value passing $\alpha!a \to 0 \parallel \alpha?x \to p$
$c_0; c_1$ has no clear correspondence
$\mathbf{if}\, gc\, \mathbf{fi}$ through $+$ and $b \to p$

Pure CCS

Eliminate CCS from natural and variable numbers

$p :=$	$\alpha.p \mid \overline{\alpha}.p \mid$	comunicazione
	$\tau.p \mid$	azione vuota
	$\Sigma_i p_i \mid$	scelta potenzialmente infinita
	$p_0 \parallel p_1 \mid$	parallelismo
	$p \backslash \alpha \mid$	restrizione
	$p[f] \mid$	relabelling
	P	identificatori di processo

One loses the distinction between output input, there remains a synchronization mechanism, α, $\overline{\alpha}$.

More abstract system.
Natural numbers and simulated guards, partly through channels and the infinite sum (extensional representation)

Trace Equivalence

- When are two processes equivalent?
- Label Transition Systems (LTS) allow more observations than standard operational semantics.
- Two processes are equivalent if they generate the same traces (deriving trees with the same paths)
- This notion of equivalence is not a congruence
- $a.(b + c)$ vs. $a.b + a.c$ in the context $\_ \parallel \overline{a}.\overline{b}$

Bisimulation

When are two processes equivalent?
When their derivative trees coincide up to duplication.

definition

- A strong bisimulation is a symmetric symmetric relation R such that:
  $p \, R \, q \ \wedge \ p \xrightarrow{\lambda} p'$
  then there exists q' tc $q \xrightarrow{\lambda} q' \ \wedge \ p' \, R \, q'$
- $p \sim q$, p is strongly bound to q if there exists a simulation R tc $p \, R \, q$

Exercise: show that $\sim$ is a bisimulation (the maximum).

Strong Bisimulation

It is a good notion of equivalence?

- Positive Aspects. It is a congruence. It is possible to reason in a compositional way.
- Negative aspects. One observes the $\xrightarrow{\tau}$ transaction, that should be an invisible action.

Weak bisimulation

definition

- A weak bisimulation is a *symmetric* R relation that:
  $p \, R \, q \;\wedge\; p \xrightarrow{a} p'$
  then there exists q', q', q'' . $\; q \xrightarrow{\tau}{}^{*} q' \xrightarrow{a} q'' \xrightarrow{\tau}{}^{*} q''' \wedge \; p' \, R \, q'''$
  and, if $p \, R \, q \;\wedge\; p \xrightarrow{\tau} p'$
  then there exists q' . $\; q \xrightarrow{\tau}{}^{*} q' \;\wedge\; p' \, R \, q'$.
- $p \sim q$, p is slightly weak at q if there exists a simulation R tc $p \, R \, q$

- Identify processes that are naturally equivalent.
- The weak bisimulation is not congruence.
  Example $\tau.a$, a, and $\tau.a + b$, $a + b$

Logic by Hennessy-Milner

A modal logic for labeled transactions. Propositions:

$$A := T \ \mid \ A_o \vee A_1 \ \mid \ \langle \lambda \rangle A \mid \neg A$$

Other connective can be defined through negation :

$$A := F \ \mid \ A_o \wedge A_1 \ \mid \ [\lambda] A$$

Satisfaction with a formula;

$$p \models A$$

We define two logically equivalent processes if they meet the same set of propositions.
Show that two strong bisimilar processes are logically equivalent.
Do the opposite?