Lecture Notes for

Monad-Based Programming

Recorded by Hans-Peter Deifel (hpd@hpdeifel.de)
Edited by Sergey Goncharov (sergey.goncharov@fau.de)

by PD Dr. Sergey Goncharov

2023/07/28

Contents

1 Semantics for Computation

In mathematics we do not distinguish between *expressions* and their *meanings*. The meaning of $2+2$ is 4 and both things mean (or *denote*) the same. In computer science we $\underline{\text{do}}$ distinguish expressions or terms from what they mean, for which we use *semantic brackets*

$$[\![-]\!]\colon \text{Terms} \to \text{Meanings}$$

The style of semantics involving such brackets is called *denotational semantics*. Denotational semantics has been developed in 70's by Christopher Strachey and Dana Scott.

The equality $2 + 2 = 4$ and the like, which we know from mathematics means that $2 + 2$ and 4 denote the same, however, what connects $2+2$ and 4 is a *computational process* (which is, of course, very simple in this case). Mathematics traditionally ignores the computational overhead of *evaluating* $2 + 2$ to 4, but in programming we cannot afford this, because programming (program analysis, verification) is largely about evaluation of expressions (or, more generally, about the process of computation). There are traces of this issue in mathematics, though, e.g. in the form of infinite series. Those usually make mathematicians uneasy, and they become much happier if they manage to find a *closed form*, i.e. an analytic expression, to which the sum converges. For examples:

$$1 + 1 + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ldots = e \approx 2.71828$$

Riemann showed that if the partial sums $\sum_{i=0}^{n} a_n$ converge, but partial sums of absolute values $\sum_{i=0}^{n} |a_n|$ diverge, then one can rearrange the elements in $\sum_{i=0}^{\infty} a_n$, so that it converges to *any* given number. Examples:

$$\begin{aligned}
(1-1) + (1/2 - 1/2) + (1/3 - 1/3) + \ldots &= 0\\
(1 + 1/2 - 1) + (1/3 + 1/4 - 1/2) + (1/5 + 1/6 + 1/7 - 1/3) + \ldots &= \ln 2\\
(1 + 1/2 - 1) + (1/3 + \ldots + 1/8 - 1/2) + (1/9 + \ldots + 1/16 - 1/4) + \ldots &= \infty
\end{aligned}$$

This makes the theory of infinite series a sophisticated subject. In computer science we deal with potentially infinite computations routinely, as we must, since Turing complete languages must express all partial recursive functions, which are those for which we generally cannot decide termination. But, on a positive side, we do not have a behaviour as sophisticated as above, which is caused by adding and subtracting quantities infinitely. The core idea of denotational semantics is that the amount of information generated with a computation keeps increasing over time, and what has been computed previous cannot be "undone" (like with the expression $1 - 1$ where -1 cancels previously computed 1).

Denotational semantics requires a system of *domains*, for collecting *values*. Classical mathematics is based on the classical set theory, which postulates that everything is a set (numbers, relations, functions, curves, etc.) Sets thus play the role of domains. In computer science, domains are chosen differently (we will see how!), as they must correctly capture the notion of partiality of data and possible non-termination of functions. Because of the parallels between mathematical functions and programs, built in the denotational semantics, the latter is sometimes called a *mathematical semantics*.

A different established style of semantics is *operational semantics*: it describes a *reduction* of terms to values, e.g. $2 + 2$ reduces to 4, assuming that $2 = s(s(0))$ (2 is the successor of the successor of 0):

$$s(s(0)) + s(s(0)) \to s(s(0) + s(s(0))) \to s(s(0 + s(s(0)))) \to s(s(s(s(0)))).$$

This neither directly defines domains of values nor identifies $+$ as a function over these values. The framework for defining operational semantics rigorously is the framework of *formal systems*. Finally, *logical semantics* describes programs by drawing on logical properties they are expected to satisfy, e.g. $x := x{+}1$ is such a program that if x was n before its execution then x is $n + 1$ after its execution.

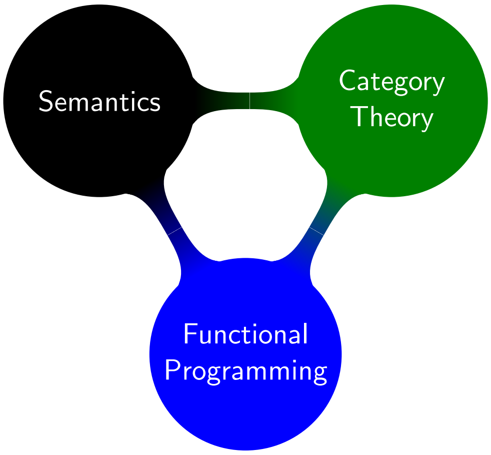
In summary:

Classical styles of semantics

- Denotational Semantics (what the program means?)
- Operational Semantics (how the program behaves?)
- Axiomatic Semantics (what properties the program satisfies?)

We stick to the first two styles of semantics, of which we first consider the second one (which is easier) to approach the first one (which is harder). Example of axiomatic semantics is *Hoare logic* (not covered here).

What we do in the course? The course revolves around the triad:



Starting from one node you will be able to connect to the other nodes, transferring the knowledge and understanding.

- Denotational semantics is motivated by computation and ultimately involves advanced mathematical structures, for which category theory is arguably the most natural language to use. We thus transfer computational intuition from semantics to category theory to approach the latter.

- Good understanding of semantics helps in functional programming, in particular Haskell, since it has been designed by computer scientists who took semantics very seriously. We thus learn Haskell in a semantic-oriented way.

- Category theory influenced semantics, since many abstract, purely mathematical concepts, such as *monads*, were utilized in semantics to organize constructions and reasoning. We thus use semantics to develop a computational intuition of formal categorical concepts.

- Similarly, a great amount of abstract categorical concepts was utilized in functional programming, again, most notably by Haskell. Specifically, monads were introduced to Haskell as a practical organization tool for writing programs – even writing the ”Hello World” program in Haskell requires a monad!

- Therefore, in this course, conversely, we use Haskell as a showcase for advanced categorical concepts, such as natural transformations, monads, adjunctions, Cartesian closure.

- Semantically, Haskell is a statically typed, purely functional lazy programming language, which can be regarded as a far-reaching generalization of the typed λ-calculus, and as such it provides as excellent playground for illustrating various important semantics concepts.

1.1 The Untyped Lambda Calculus

Untyped λ-calculus is a proto-programming language introduced by a mathematician *Alonzo Church* in 1930’s prior to any actual programming languages and computers. We proceed to recall some general facts about the λ-calculus.

$$\begin{aligned} \text{Variables} \quad & x, y, z, \ldots \\ \text{Terms} \quad & t, s ::= x, y, z \mid \lambda x.\, t \mid ts \\ \text{Contexts} \quad & C ::= \Box \mid \lambda x.\, C \mid Ct \mid tC \end{aligned}$$

So, a context, more precisely, a *linear context*, is a term with one “hole” $\Box$. Let $C[t]$ be the term obtained by replacing $\Box$ in a context C with a term t.

- α-conversion $C[\lambda x.\, t] \longrightarrow_\alpha C[\lambda y.\, t[y/x]]$, where y is bound in t (see definition below)
- β-reduction $C[(\lambda x.\, t)s] \longrightarrow_\beta C[t[s/x]]$
- η-reduction $C[\lambda x. fx] \longrightarrow_\eta C[f]$

where C ranges over all contexts. Derived reductions:

- $\alpha\beta$-reduction is: $\longrightarrow^\star_{\alpha\beta} = (\to_\alpha \cup \to_\beta)^\star$
- $\alpha\beta\eta$-reduction is: $\longrightarrow^\star_{\alpha\beta\eta} = (\to_\alpha \cup \to_\beta \cup \to_\eta)^\star$

Definition (Redex)**.** A ($\beta-$)*redex (=reducible expression)* is a subterm of the form $(\lambda x.\, t)s$ of a given term; that is, the given term is of the form $C[\lambda x.\, t]$.

Definition (Free Variables)**.**

- $\mathrm{Free}(x) = \{x\}$
- $\mathrm{Free}(st) = \mathrm{Free}(s) \cup \mathrm{Free}(t)$

- $\mathrm{Free}(\lambda x.s) = \mathrm{Free}(s) \smallsetminus \{x\}$

A variable x is *free* in t, if $x \in \mathrm{Free}(t)$. A variable x is *bound* in t, if $x \notin \mathrm{Free}(t)$.

Definition (Substitution)**.**

- $x[t/x] = t$;
- $x[t/y] = x$ if $x \neq y$;
- $(pq)[t/x] = p[t/x]q[t/x]$;
- $(\lambda x.\, p)[t/x] = \lambda x.\, p$;
- $(\lambda y.\, p)[t/x] = \lambda z.\, p[z/y][t/x]$ if $z \notin \mathrm{Free}(\lambda y.\, p) \cup \mathrm{Free}(t)$.

Example. $(\lambda x.\, yx)[yx/y] = \lambda z.(yx)[z/x][yx/y] = \lambda z.(yz)[yx/y] = \lambda z.(yx)z.$

Proposition (Diamond Property = Confluence = Church-Rosser Property)**.** Independent reductions starting from the same term can always eventually be joined in the following sense:

$$\begin{tikzcd}
 & t \arrow[dl, "\alpha\beta"', "*"] \arrow[dr, "\alpha\beta", "*"'] & \\
t_1 \arrow[dr, dashed, "\alpha\beta"', "*"] & & t_2 \arrow[dl, dashed, "\alpha\beta", "*"'] \\
 & s &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & t \arrow[dl, "\alpha\beta\eta"', "*"] \arrow[dr, "\alpha\beta\eta", "*"'] & \\
t_1 \arrow[dr, dashed, "\alpha\beta\eta"', "*"] & & t_2 \arrow[dl, dashed, "\alpha\beta\eta", "*"'] \\
 & s &
\end{tikzcd}$$

That identifies one-step relations $\to_{\alpha\beta}$ and $\to_{\alpha\beta\eta}$ as confluent, or Church-Rosser and their transitive-reflexive closures $\to^\star_{\alpha\beta}$ and $\to^\star_{\alpha\beta\eta}$ as having the diamond property.

The equivalent formulation of the Church-Rosser property, say for $\to_{\alpha\beta}$, is as follows: $s(\to_{\alpha\beta} \cup_{\alpha\beta} \leftarrow)^\star t$ iff there is r such that $s \to^\star_{\alpha\beta} r$ and $t \to^\star_{\alpha\beta} r$. In other words, s and t are reachable from each other by zig-zaging with $\to_{\alpha\beta}$ back and forth iff there is a term r, to which both s and t reduce by $\to_{\alpha\beta}$.

Proposition. $\longrightarrow^\star_{\alpha\beta}$ is not terminating:

Proof. Since $\Omega = (\lambda x.\, xx)(\lambda x.\, xx) \longrightarrow_\beta (\lambda x.\, xx)(\lambda x.\, xx) = \Omega$, we obtain and infinite reduction $\Omega \to_\beta \Omega \to_\beta \dots$ □

It follows that $t \longrightarrow_\alpha s$ iff $s \longrightarrow_\alpha t$, which entails that $\longrightarrow^\star_\alpha$ is an *equivalence relation.* Following the usual approach, we will dim the distinction between α-equivalent terms. The slogan is: terms that are equal up to renaming bound variables are considered to be equal; one also says: equal *up-to α-equivalence.*

Definition (Fixpoint Combinator)**.** $Y = \lambda f.\, (\lambda x.\, f(xx))(\lambda x.\, f(xx))$

$$Yf \to_\beta (\lambda x.\, f(xx))(\lambda x.\, f(xx)) \to_\beta f((\lambda x.\, f(xx))(\lambda x.\, f(xx)))\ {}_\beta\!\leftarrow f(Yf),$$

so Yf and $f(Yf)$ are β-equivalent, but Yf need not β-reduce to $f(Yf)$.

Definition (Church Numerals)**.** The following combinators model natural numbers:

$$\begin{aligned}\underline{0} &= \lambda f.\, \lambda z.z\\ \underline{1} &= \lambda f.\, \lambda z.fz\\ \underline{2} &= \lambda f.\, \lambda z.ffz\\ &\vdots\end{aligned}$$

They can be added with

$$(+) = \lambda m.\, \lambda n.\, \lambda f.\, \lambda z.\, m\ f\ (n\ f\ z)$$

In a similar way one can define $(-)$, True, False, *if-then-else*, etc.

Untyped λ-calculus is a precursor of functional programming languages. It is elegant from a purely mathematical perspective, but from the practical programming point of view it is subject to serious drawbacks in its core.

Benefits:

- Confluent, hence equality of programs arises from β-reduction, which is a basic notion of program interpretation.
- Turing complete, as indicated by non-termination.
- Higher-order from the outset, so functions can be passed around, just like data.

Shortcomings:

- No concrete mechanism for preferring one β-reduction over another, although the choice can be vital for performance.
- The assumption that an interpreter would deeply inspect the program for potential redexes is unrealistic.
- No distinction between closed terms and terms with free variables, hence no notion of a *value* as a complete closed and irreducible piece of data.
- Definable constructs generally fail to be subject to expected reductions, and might satisfy only unoriented β-equivalences instead, e.g. $(Y f)\, p \to^{\star}_{\beta} \cdot\ {}_{\beta}\!\leftarrow f\ (Y f) p$ instead of $(Y f)\, p \to^{\star}_{\beta} f (Y f)\, p$.

1.2 Evaluation Strategies

Evaluation strategies describe how a term can be reduced. In particular, we might want an evaluation strategy to be *deterministic*, since an implementation of it in a compiler must be so. An appropriate language for defining evaluation strategies (deterministic or not) is the language of formal systems.

1.2.1 Formal Systems

Formal systems is a language of mathematics and (!) of theoretical computer science. They describe, how *new* pieces of knowledge can be obtained from *old* in a rule-based manner from top to bottom, by building a finitary derivation where we move from assumptions (or facts) to goals.

Definition (Formal System)**.** A formal system consists of

- A (finite) set of symbols – *alphabet*;
- A *grammar* for producing *formulas* from symbols. A formula is said to be well-formed if it can be formed using the rules of the grammar. Since one is usually not interested in non-well-formed formulas for too long, one usually shortens “well-formed formula” to “formula”;
- A set of *axioms*, or axiom schemata, consisting of well-formed formulas;
- A set of *inference rules*, consisting of multiple (zero or infinite number of) premises and precisely one conclusion, depicted as

  $$\frac{\Phi_1 \quad \Phi_2 \quad \dots \quad \Phi_n}{\Phi}$$

  if the number of premises $(\Phi_1, \dots, \Phi_n)$ is finite; here Φ is the conclusion.

Derivations are built by connecting conclusions of rule instances with premises of rule instances in acyclic manner. We only accept derivations which are *complete* (no pending global premises) and *well-founded* (every path of the derivation, viewed as a tree, is finite). If every rule of the formal system has only finitely many premises, a derivation is well-founded iff it is finite (*Kőnig’s lemma*). A formula is *derivable* if it can be a net conclusion of some derivation.

Some remarks:

- One is usually interested in organizing a formal system in some sort of finitary (technically speaking: recursively enumerable) way. That is, if there is a finite number of axioms and rules, we are fine. Otherwise, we might need to capture many axioms and rules with *schemata*, meaning that even thought, the number can be infinite, but there is a computationally meaningful procedure to enumerate them all.
- An axiom is virtually a rule with no premises.
- Aside from the logical context, it can be more suitable to call formulas *judgements*, meaning that a judgement is something more general than a formula. Derivable formulas are also called *theorems*, but that again only makes sense if judgements are some sort of logical formulas, which are true or falls. Formal systems, generally speaking, operate with derivable (or not derivable), and not just with true or false.

Example (Cherry-Banana Calculus)**.** Let $\{🍒, 🍌\}$ be the alphabet, and let the grammar identify any non-empty sequence over $\{🍒, 🍌\}$ as a (well-formed) formula. Rule schemes

$$\frac{}{🍒}\ (\mathrm{i}) \qquad\qquad \frac{x}{🍌\, x}\ (\mathrm{ii}) \qquad\qquad \frac{x\,🍌 \qquad 🍌\, y}{xy}\ (\mathrm{iii}) \qquad\qquad \frac{x}{🍒\, x\, 🍌}\ (\mathrm{iv})$$

represent an infinite number of *rules*, obtained by replacing *variables* x, y with non-empty finite sequences of 🍒 and 🍌. Rule (i) is an “axiom”.

We can build *proofs* or *derivations*, like

$$\dfrac{\dfrac{\dfrac{}{\text{🍒}}\,(i)}{\text{🍒 🍒 🍌}}\,(iv) \qquad \dfrac{\dfrac{}{\text{🍒}}\,(i)}{\text{🍌 🍒}}\,(ii)}{\text{🍒 🍒 🍒}}\,(iii)$$

Thus the formula 🍒 🍒 🍒 is derivable.

Contrastingly, let us show that 🍒 🍌 is *not* derivable. Indeed, if it was derivable, it would be derivable with rule (iii). But that rule itself would require 🍒 🍌 as a premise – we obtain contradiction to the global assumption that derivations must be finite.

Example (Transitive-Reflexive Closure)**.** Given a set X and a relation $R \subseteq X \times X$, we previously used the transitive-reflexive closure $R^\star \subseteq X \times X$ of R. A formal way to define $R^\star$ is by describing a corresponding formal system:

- alphabet: elements of X and $R^\star$, regarded as a formula, disjoint from X;
- formulas: $xR^\star y$;
- rules:

$$\frac{}{xR^\star x} \qquad\qquad \frac{}{xR^\star y}\quad \text{for such } x,\ y \text{ that } xRy \qquad\qquad \frac{xR^\star y \quad yR^\star z}{xR^\star z}$$

Note that natural deduction (from GLoIn) cannot be organized as a formal system so easily, e.g. it has rules like

$$\dfrac{\begin{array}{|l} \phi \\ \hline \vdots \\ \psi \end{array}}{\phi \Rightarrow \psi}\ (\Rightarrow \mathrm{I})$$

That is: derivations themselves must be judgements. Gentzen solved that by introduced a *sequent calculus* for first-order logic, whose judgements are *sequents*

$$\phi_1, \dots, \phi_n \vdash \psi_1, \dots, \psi_m$$

where the ϕ_i are first-order formulas (conjunctive premises) and the ψ_j are again first-order formulas (disjunctive goals).

Formal systems are perfectly suitable for describing program semantics: when judging that a program (terminates and) returns a value, it is natural to assume that this is something we can derive with a finitary system of rules in finitely many steps (in contrast to judging that a program *does not* terminate, which need not be derivable in finitely many steps).

1.2.2 Standard Evaluation Strategy

We specify evaluation strategies with rules of *structural operational semantics (SOS)*. SOS is a class of formal systems where the judgments describe how programs reduce. “Structural” means, that the premises for a judgement on how to reduce a program, are judgements about reducing structurally smaller programs. We proceed with the *small-step* operational semantics where the judgements have the form $s \to s'$ meaning that s reduces to s' in one step.

The evaluation order imposed by the standard evaluation strategy is called the *left-most-outermost order*.

$$\frac{}{(\lambda x.\, p)q \longrightarrow_{\text{so}} p[q/x]} \qquad \frac{p \longrightarrow_{\text{so}} p' \qquad p \neq \lambda y.\, t}{pq \longrightarrow_{\text{so}} p'q} \qquad \frac{p \longrightarrow_{\text{so}} p'}{\lambda x.\, p \longrightarrow_{\text{so}} \lambda x.\, p'}$$

$$\frac{q \longrightarrow_{\text{so}} q' \qquad p \downarrow_{\text{so}} \qquad p \neq \lambda x.\, t}{pq \longrightarrow_{\text{so}} pq'}$$

where $p \downarrow_{\text{so}}$ means that p is irreducible with respect to $\longrightarrow_{\text{so}}$, i.e. p is so-normal.

Remark. The auxiliary judgements like $p \neq \lambda x.\, t$ and $p \downarrow_{\text{so}}$ are, strictly speaking, side-conditions, as they do not take part in the derivation process. The latter one, though, indirectly refers to it, by stating that no derivation $p \longrightarrow_{\text{so}} p'$ exists. This is an example of so-called *negative premise*. Those can potentially be a problem. In our case, one can show that $p \downarrow_{\text{so}}$ iff p is β-normal, and thus we can get rid of this undesirable circularity.

This style of reductions is also called *small-step semantics* because in order to find an so-normal form p' of some p we generally need a chain of reductions $p \longrightarrow_{\text{so}} \dots \longrightarrow_{\text{so}} p'$.

Definition. Using these rules, we define $p \downarrow_{\text{so}} v$, if there is a derivation of $p \longrightarrow^{\star}_{\text{so}} v$ and v is so-normal.

Example.

$$\frac{\dfrac{}{(\lambda x.\, xy)(\lambda x.\, x) \longrightarrow_{\text{so}} (\lambda x.\, x)y} \qquad y \downarrow_{\text{so}} \qquad y \neq \lambda x.\, t}{y((\lambda x.\, xy)(\lambda x.\, x)) \longrightarrow_{\text{so}} y((\lambda x.\, x)y)}$$

Proposition (Standardization Theorem[1])**.** If $s \longrightarrow^{\star}_{\alpha\beta} t$ and t is β-normal, then $s \longrightarrow^{\star}_{\text{so}} t$ and t is so-normal.

Note the following.

- The definition of $\longrightarrow_{\text{so}}$ is *structural*, i.e. a successor of a term t w.r.t. $\longrightarrow_{\text{so}}$ is calculated by structural induction over t.
- The relation $\longrightarrow_{\text{so}}$ is *deterministic* in the sense that there is only one way to build a (possibly nonterminating) reduction starting from a given t; this contrasts $\alpha\beta$-reduction: we both have $(\lambda x.\lambda y.\, y)\Omega \longrightarrow_{\beta} \lambda y.\, y$ and

$$(\lambda x.\lambda y.\, y)\Omega \longrightarrow_{\text{so}} (\lambda x.\lambda y.\, y)\Omega \longrightarrow_{\text{so}} \cdots$$

[1] Hendrik Barendregt. *The Lambda calculus: Its syntax and semantics.* Amsterdam: North-Holland, 1984, but see ThProg for a beautiful and concise proof!

- The standartization theorem indicates that all existing β-normal forms can be calculated by the standard evaluation, e.g. $(\lambda x.\lambda y.\, y)\,\Omega \longrightarrow_{\text{so}} \lambda y.\, y$ and $\lambda y.\, y \downarrow_{\text{so}}$.

- As a consequence of the previous clause, $\longrightarrow_{\text{so}}$ diverges on a term t iff t does not have an β-normal form.

1.2.3 Call-by-Name (Lazy) Evaluation Strategy

Lazy or *call-by-name (CBN)* evaluation strategy refines and simplifies the standard evaluation strategy as follows:

$$\frac{}{(\lambda x.\, p)q \longrightarrow_{\text{cbn}} p[q/x]} \qquad\qquad \frac{p \longrightarrow_{\text{cbn}} p'}{pq \longrightarrow_{\text{cbn}} p'q}$$

where the terms are now supposed to be closed. Compared to the standard evaluation strategy, the key distinctions are:

- no more rewriting under λ (therefore $\lambda x.\, \Omega \downarrow_{cbn}$);
- all terms are closed.

We now explicitly reject η-reduction, in order to capture the fundamental distinction between *computations* and *values*. Roughly, a λ-term p represents a program, and $\lambda x.\, px$ represents its program code. While p can diverge, $\lambda x.\, p$ cannot diverge, because it is just a text of the program. However $\lambda x.\, p$ can be applied to an argument, which then can again result in divergence.

Proposition. Like SO, CBN does not diverge on terms which have β-normal forms, but CBN-normal forms need not be β-normal, e.g. $\lambda x.\ (\lambda y.\, y)x \downarrow_{cbn}$ but $\lambda x.\, (\lambda y.\, y)x \to_\beta \lambda x.\, x$.

Example.

$$\begin{aligned}
(\lambda x.\, xx)((\lambda x.\, x)(\lambda x.\, x)) &\longrightarrow_{\text{cbn}} ((\lambda x.\, x)(\lambda x.\, x))((\lambda x.\, x)(\lambda x.\, x)) \\
&\longrightarrow_{\text{cbn}} (\lambda x.\, x)((\lambda x.\, x)(\lambda x.\, x)) \\
&\longrightarrow_{\text{cbn}} (\lambda x.\, x)(\lambda x.\, x) \\
&\longrightarrow_{\text{cbn}} (\lambda x.\, x).
\end{aligned}$$

1.2.4 Call-by-Value (Eager) Evaluation Strategy

Definition (Value)**.** A value is a term of the form $\lambda x.\, t$.

Under the same assumption as with CBN we define the *call-by-value (CBV)* evaluation strategy:

$$\frac{p \longrightarrow_{\text{cbv}} p'}{pq \longrightarrow_{\text{cbv}} p'q} \qquad \frac{q \longrightarrow_{\text{cbv}} q' \qquad p\ \text{ is a value}}{pq \longrightarrow_{\text{cbv}} pq'} \qquad \frac{q\ \text{ is a value}}{(\lambda x.\, p)q \longrightarrow_{\text{cbv}} p[q/x]}$$

instead of "p is a value", one could write $p \downarrow_{\text{cbv}}$.

Proposition. CBV calculates *properly fewer* normal forms than CBN, e.g. $(\lambda x.\lambda y.\,y)\Omega \downarrow_{cbn} \lambda y.\,y$, but

$$(\lambda x.\lambda y.\,y)\,\Omega \longrightarrow_{\text{cbv}} (\lambda x.\lambda y.\,y)\,\Omega \longrightarrow_{\text{cbv}} \cdots$$

However, CBV is generally more efficient than CBN.

Example. We can redo the previous example in the CBV style:

$$\begin{aligned}(\lambda x.\,xx)((\lambda x.\,x)(\lambda x.\,x)) &\longrightarrow_{\text{cbv}} (\lambda x.\,xx)(\lambda x.\,x)\\ &\longrightarrow_{\text{cbv}} (\lambda x.\,x)(\lambda x.\,x)\\ &\longrightarrow_{\text{cbv}} (\lambda x.\,x).\end{aligned}$$

This demonstrates that CBV (at least, implemented naively) is more efficient than CBN.

1.2.5 Big-Step Call-by-Name

In big-step styles of semantics we relate a term not to its one-step successor, but directly to its normal form.

$$\frac{}{\lambda x.\,p \Downarrow_{\text{cbn}} \lambda x.\,p} \qquad\qquad \frac{p \Downarrow_{\text{cbn}} \lambda x.\,p' \qquad p'[q/x] \Downarrow_{\text{cbn}} c}{pq \Downarrow_{\text{cbn}} c}$$

Proposition. $p \longrightarrow^{\star}_{\text{cbn}} q$ and $q \downarrow_{\text{cbn}}$ iff $p \Downarrow_{\text{cbn}} q$.

Proving this requires the following

Lemma. $p \longrightarrow_{\text{cbn}} q$ with $q \Downarrow_{\text{cbn}} r$ imply $p \Downarrow_{\text{cbn}} r$.

Proof. Induction over the proof of $p \longrightarrow_{\text{cbn}} q$:
Induction base: $p = (\lambda x.\,t)s$, $q = t[s/x]$. Then we build the necessary derivation $p \Downarrow_{\text{cbn}} r$ in two steps.
Induction step: $p = st$, $q = s't$ and $s \longrightarrow_{\text{cbn}} s'$. By assumption, $s't \Downarrow_{\text{cbn}} r$, which implies $s' \Downarrow_{\text{cbn}} \lambda x.\,u$, $u[t/x] \Downarrow_{\text{cbn}} r$. By induction, $s \Downarrow_{\text{cbn}} \lambda x.\,u$. Hence $st \Downarrow_{\text{cbn}} r$, as required. □

1.2.6 Big-Step Call-by-Value

Call-by-value requires evaluation of arguments of function application:

$$\frac{}{\lambda x.\,p \Downarrow_{\text{cbv}} \lambda x.\,p} \qquad\qquad \frac{p \Downarrow_{\text{cbv}} \lambda x.p' \qquad q \Downarrow_{\text{cbv}} q' \qquad p'[q'/x] \Downarrow_{\text{cbv}} c}{pq \Downarrow_{\text{cbv}} c}$$

Proposition. $p \longrightarrow^{\star}_{\text{cbv}} q$ and $q \downarrow_{\text{cbv}}$ iff $p \Downarrow_{\text{cbv}} q$.

Example.

$$\dfrac{\dfrac{}{\lambda x.\,xx \Downarrow_{\text{cbv}} \lambda x.\,xx} \qquad \dfrac{\dfrac{}{\lambda x.\,x \Downarrow_{\text{cbv}} \lambda x.\,x} \quad \dfrac{}{\lambda x.\,x \Downarrow_{\text{cbv}} \lambda x.\,x} \quad \dfrac{}{\lambda x.\,x \Downarrow_{\text{cbv}} \lambda x.\,x}}{(\lambda x.\,x)(\lambda x.\,x) \Downarrow_{\text{cbv}} \lambda x.\,x} \qquad \dfrac{}{(\lambda x.\,x)(\lambda x.\,x) \Downarrow_{\text{cbv}} \lambda x.\,x}}{(\lambda x.\,xx)((\lambda x.\,x)(\lambda x.\,x)) \Downarrow_{\text{cbv}} \lambda x.\,x}$$

1.3 PCF (Programming Computable Functions)

1.3.1 Simply-Typed λ-calculus

$$\text{Type} ::= \underbrace{A, B, C, \ldots}_{\text{base types}} \mid \underbrace{1}_{\text{unit type}} \mid \underbrace{A \times B}_{\text{product types}} \mid \underbrace{A \to B}_{\text{function types}}$$

Proposition. $\Omega = (\lambda x.\, xx)(\lambda x.\, xx)$ is not typable, and hence not a valid term.

Proof. By contradiction: if $x\colon A$ then $xx\colon A$ and $x\colon A \to B$, hence $A = A \to B$, contradiction. □

Proposition. $\longrightarrow_\beta$ is strong normalising for simply typed λ-calculus.

PCF is obtained from the simply typed λ-calculus by

- fixing *Nat* and *Bool* as the base types;
- postulating the corresponding signature of arithmetic and logical operations;
- adding if-then-else;
- adding the fixpoint combinator $Y_A\colon (A \to A) \to A$; for every type A.

Definition (Terms-In-Context)**.** A *term in context* has the form

$$\Gamma \vdash t\colon A,$$

where A is a type and Γ is a context, which is a list of pairs $x_i\colon A_i$ such that x_i occur non-repetitively.

We work only with those $\Gamma \vdash t\colon A$ which are derivable using the following rules:

$$\textbf{(Var)}\ \frac{x\colon A \quad \text{is in } \Gamma}{\Gamma \vdash x\colon A} \qquad \textbf{(1I)}\ \frac{}{\Gamma \vdash \star\colon 1} \qquad (\times\ \textbf{I})\ \frac{\Gamma \vdash t\colon A \qquad \Gamma \vdash s\colon B}{\Gamma \vdash \langle t, s\rangle\colon A \times B}$$

$$(\times\ \textbf{E}_1)\ \frac{\Gamma \vdash t\colon A \times B}{\Gamma \vdash \mathsf{fst}\, t\colon A} \qquad (\times\ \textbf{E}_2)\ \frac{\Gamma \vdash t\colon A \times B}{\Gamma \vdash \mathsf{snd}\, t\colon B}$$

$$(\to \textbf{I})\ \frac{\Gamma, x\colon A \vdash t\colon B}{\Gamma \vdash \lambda x.\, t\colon A \to B} \qquad (\to \textbf{E})\ \frac{\Gamma \vdash s\colon A \to B \qquad \Gamma \vdash t\colon A}{\Gamma \vdash st\colon B}$$

$$\textbf{(Const)}\ \frac{}{\Gamma \vdash c\colon A} \qquad \textbf{(Fun)}\ \frac{\Gamma \vdash t_1\colon A_1 \quad \cdots \quad \Gamma \vdash t_n\colon A_n}{\Gamma \vdash f(t_1, \ldots, t_n)\colon B}$$

where $c \in \{\mathsf{True}, \mathsf{False}\} \cup \{0, 1, \ldots\}$ where $f \in \{\wedge, \vee, \neg, +, -, \ldots\}$

$$\textbf{(Eq)}\ \frac{\Gamma \vdash s\colon A \qquad \Gamma \vdash t\colon A \qquad A \in \{\mathit{Bool}, \mathit{Nat}, 1\}}{\Gamma \vdash s = t\colon \mathit{Bool}}$$

$$\textbf{(If)}\ \frac{\Gamma \vdash b\colon \mathit{Bool} \qquad \Gamma \vdash s\colon A \qquad \Gamma \vdash t\colon A}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t\colon A} \qquad \textbf{(Fix)}\ \frac{}{\Gamma \vdash Y_A\colon (A \to A) \to A}$$

Definition (Term)**.** A PCF *term* t is obtained from $\Gamma \vdash t\colon A$ by removing the return type A and the context Γ.

The PCF syntax corresponds to the Haskell syntax quite accurately, e.g.:

```haskell
-- | single element () of the unit type ()
() :: ()

-- | first component of a pair
fst                             :: (a,b) -> a
fst (x,_)                       =  x

-- | second component of a pair
snd                             :: (a,b) -> b
snd (_,y)                       =  y

-- | logical constants
True                            :: Bool
False                           :: Bool

-- | Numeric constants
0                               :: Integer
42                              :: Integer

-- | lambda-abstraction, assuming f :: a -> b
\x -> f x                       :: a -> b

-- | application, assuming f :: a -> b, x :: a
f x                             :: b

-- | equality
(==)                            :: Eq a => a -> a -> Bool

-- | if-then-else, assuming b :: Bool, x :: a, y :: a
if b then a else b              :: a

-- | fixpoint operator is definable:
fix                             :: (a -> a) -> a
fix f                           = f(fix f)
```

1.3.2 Call-by-Name Operational Semantics for PCF

We modify the concept of value as follows.

Definition (Value)**.** A value is either

- a Boolean, or

- a natural number, or
- $\star$, or
- a pair of closed terms, or
- a closed term $\lambda x.\, t$.

The call-by-name semantics for PCF is obtained by completing the call-by-name semantics of λ-calculus. As before, the judgement $p \Downarrow v$ indicates that p reduced to the value v in the updated sense.

We discuss the most instructive rules.

$$\frac{t \Downarrow \langle p, q\rangle \qquad p \Downarrow c}{\mathsf{fst}\, t \Downarrow c} \qquad\qquad \frac{t \Downarrow \langle p, q\rangle \qquad q \Downarrow c}{\mathsf{snd}\, t \Downarrow c}$$

which means that pairing is lazy. Hence, in particular, $\mathsf{fst}\langle 1, \Omega\rangle \Downarrow 1$, but $\mathsf{snd}\langle 1, \Omega\rangle$ diverges. Note that there is no rule for reducing $\langle t, s\rangle$, which is by definition already a value.

$$\frac{b \Downarrow \mathsf{True} \qquad p \Downarrow c}{\mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else}\ q \Downarrow c} \qquad\qquad \frac{b \Downarrow \mathsf{False} \qquad q \Downarrow c}{\mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else}\ q \Downarrow c}$$

The rules for application and abstraction are as in the λ-calculus. Arithmetic operations are strict (i.e. if one argument fails, everything fails):

$$\frac{p \Downarrow c_1 \qquad q \Downarrow c_2}{p + q \Downarrow c_1 + c_2}$$

For interpreting logical disjunction, one could think of the following seemingly natural rules:

$$\frac{b \Downarrow \mathsf{True}}{b \vee c \Downarrow \mathsf{True}} \qquad\qquad \frac{c \Downarrow \mathsf{True}}{b \vee c \Downarrow \mathsf{True}} \qquad\qquad \frac{b \Downarrow \mathsf{False} \qquad c \Downarrow \mathsf{False}}{b \vee c \Downarrow \mathsf{False}}$$

This is known as *"parallel or"* and it does make certain sense, but in our case it would make the semantics unintentionally non-deterministic. That is, to evaluate $b \vee c$, the semantics every time would need to simulate behaviours of two independent threads running in parallel and correspondingly evaluating b and c until one of them possibly succeeds. Such parallel facilities are not considered to be part of the core in functional languages. From a foundational point of view, PCF was developed for programming computable functions, in the sense of Turing computability. This notion of computability is sequential by definition, and thus also does not support facilities for parallel execution.

The standard rules are like this

$$\frac{b \Downarrow \mathsf{True}}{b \vee c \Downarrow \mathsf{True}} \qquad\qquad \frac{b \Downarrow \mathsf{False} \qquad c \Downarrow d}{b \vee c \Downarrow d}$$

That is, $\vee$ is not commutative, e.g. $\mathsf{True} \vee \Omega \Downarrow \mathsf{True}$, but $\Omega \vee \mathsf{True}$ diverges. It is easy to see that $b \vee c$ is interpreted in the same way as $\mathsf{if}\, b\, \mathsf{then}\ \mathsf{True}\ \mathsf{else}\ c$.

This semantics can be readily tested in Haskell, since it is lazy:

```
fix f   = f (fix f)                          -- fixpoint combinator
omega   = fix id                             -- divergence
success = ()                                 -- successful termination

test1 = fst   (success, omega)               -- terminates
test2 = snd   (success, omega)               -- diverges

test3 = True  || omega                       -- terminates
test4 = omega || True                        -- diverges
test4 = False || omega                       -- diverges
```

The rule for the fixpoint combinator is the only non-structural rule:

$$\frac{f(Y_A f) \Downarrow c}{Y_A f \Downarrow c}$$

1.3.3 Call-by-Value Operation Semantics for PCF

We redefine the notion of value once again.

Definition (Value)**.** A value is a Boolean, or a natural number, or $\star$, or a pair of values or a closed term $\lambda x.\, t$.

$$\frac{p \Downarrow_{\text{cbv}} c_1 \qquad q \Downarrow_{\text{cbv}} c_2}{\langle p, q\rangle \Downarrow_{\text{cbv}} \langle c_1, c_2\rangle} \qquad\qquad \frac{p \Downarrow_{\text{cbv}} \langle c_1, c_2\rangle}{\mathsf{fst}\, p \Downarrow_{\text{cbv}} c_1} \qquad\qquad \frac{p \Downarrow_{\text{cbv}} \langle c_1, c_2\rangle}{\mathsf{snd}\, p \Downarrow_{\text{cbv}} c_2}$$

If we used the same rule for the Y-combinator, as for call-by-name, we would diverge:

$$Yf \longrightarrow f(Yf) \longrightarrow f(f(Yf)) \longrightarrow \cdots$$

(Evaluating the argument would use the same rule on and on). In order to prevent this, for the CBV semantics:

- we require C in Y_C to be of the form $A \to B$,
- the small-step rule for Y: $Yf \to f(\lambda x.(Yf)x)$, or, alternatively, as a big-step rule:

$$\frac{f \Downarrow_{\text{cbv}} \lambda x.\, g \qquad g[\lambda y.\,(Yf)y/x] \Downarrow_{\text{cbv}} c}{Yf \Downarrow_{\text{cbv}} c}$$

Example (Factorial)**.** The factorial function

$$\mathsf{fac}(0) = \mathsf{fac}(1) = 1 \qquad\qquad \mathsf{fac}(n) = n \cdot \mathsf{fac}(n-1) \qquad (n > 1)$$

is expressible as follows in PCF:

$$p := \left(x \colon \mathit{Nat} \vdash (Y_{\mathit{Nat}\to\mathit{Nat}}(\underbrace{\lambda f.\lambda x.\, \mathsf{if}\ x \leqslant 1\ \mathsf{then}\ 1\ \mathsf{else}\ x \cdot f(x-1)}_{g}))(x) \right)$$

We show that: $(\lambda x.\, p)(n) \Downarrow n!$ (in CBV).

Proof. We easily reduce the goal to $(Yg)n \Downarrow n!$. We proceed by induction over n.

- If $n = 0$, then

$$\dfrac{\dfrac{\dfrac{}{g \Downarrow g} \quad \dfrac{}{\lambda x.\, \underbrace{\mathsf{if}\; x \leqslant 1\; \mathsf{then}\; 1\; \mathsf{else}\; x \cdot (\lambda y.\, (Yg)y)(x-1)}_{g'} \Downarrow \lambda x.\, g'}}{Yg \Downarrow \lambda x.\, g'} \quad \dfrac{}{0 \Downarrow 0} \quad \dfrac{\dfrac{}{0 \leqslant 1 \Downarrow \mathsf{True}} \quad \dfrac{}{1 \Downarrow 1}}{g'[0/x] \Downarrow 1}}{(Yg)0 \Downarrow 1}$$

- Case $n = 1$ is analogous.
- Suppose, $n > 0$. Then

$$\dfrac{\dfrac{\dfrac{}{g \Downarrow g} \quad \dfrac{}{\lambda x. g' \Downarrow \lambda x.\, g'}}{Yg \Downarrow \lambda x.\, g'} \quad \dfrac{}{n \Downarrow n} \quad \dfrac{\dfrac{}{n \leqslant 1 \Downarrow \mathsf{False}} \quad \dfrac{\dfrac{}{n \Downarrow n} \quad \dfrac{\dots \quad \dfrac{\vdots}{(Yg)(n-1) \Downarrow (n-1)!}}{(\lambda y.\, (Yg)y)(n-1) \Downarrow (n-1)!}}{n \cdot (\lambda y.\, (Yg)y)(n-1) \Downarrow n!}}{g'[n/x] \Downarrow n!}}{(Yg)(n) \Downarrow n!}$$

where $\vdots$ refers to the proof from the induction hypothesis. □

1.3.4 Contextual Equivalence

Operational semantics culminates in defining *contextual* (or *operational*, or *observational*) equivalence. We want to know if two programs are equivalent or not w.r.t. their observable behaviour. What this means? For closed programs, we can just compare if they reduce to the same value, i.e. s and t are equivalent iff $s \Downarrow v$ iff $t \Downarrow v$ for some v. This is not enough for programs with free variables. Intuitively, free variables range over program inputs and two programs must be equivalent if they agree on all possible inputs. Additionally, if the result type is itself a functional type, we need to check if the results are equivalent as functions. We can formally capture the relevant notion with *term contexts.*

A term context is a term with a special symbol $\Box$, occurring precisely once. Given a context C and a term t, $C[t]$ denotes the term obtained by replacing $\Box$ in C with t. In typed settings, contexts and placeholders $\Box$ have types, i.e. in PCF $(\Box + 1)$ has the type *Nat* and $\Box$ in it has the type *Nat.*

Definition (Contextual Equivalence for PCF)**.** A term context C is of *ground type* if its type is either *Nat* or *Bool* or 1. Two PCF terms $\Gamma \vdash s\colon A$ and $\Gamma \vdash t\colon A$ are *contextually equivalent* if for every context C of ground type, for every value v, $C[s] \Downarrow v$ iff $C[t] \Downarrow v$. In this case, we write $\Gamma \vdash p =_{\mathsf{ctx}} q\colon A$, or simply $p =_{\mathsf{ctx}} q$.

We see that contextual equivalence is a derived notion, defined indirectly via semantics of closed programs and by quantification over (infinitely many) term contexts. This complex notion is needed, to cope with (a) programs with free variables (b) higher-order programs (i.e. not only programs of ground types). Clearly (a) reduces to (b) by λ-abstracting free variables. The following can be shown (although, we currently do not have machinery for proving even such innocently looking statements):

Proposition. Let $\Gamma \vdash s\colon A$ and $\Gamma \vdash t\colon A$ be two programs in context, A be a ground type and let Γ be empty. Then $p =_{\mathsf{ctx}} q$ iff for every value v, $s \Downarrow v$ iff $t \Downarrow v$.

Contextual equivalence of higher-order programs is considerably more advanced than contextual equivalence of closed programs of ground types. For example, for proving that two implementations of binary addition of numbers are equivalent, it is necessary to check that they behave the same on all inputs (i.e. either both converge to the same result, or both diverge). This quantification "over all inputs" is what is more abstractly captured by quantification over all term contexts.

1.3.5 Coproducts, Abrupt Termination and I/O

Let us extend PCF with coproducts types. That is, we add $A + B$ to the type grammar and add the following term formation rules:

$$(\mathbf{+I}_1)\quad \frac{\Gamma \vdash t\colon A}{\Gamma \vdash \mathsf{inl}\, t\colon A + B} \qquad\qquad (\mathbf{+I}_2)\quad \frac{\Gamma \vdash t\colon B}{\Gamma \vdash \mathsf{inr}\, t\colon A + B}$$

$$(\mathbf{+E})\quad \frac{\Gamma \vdash b\colon A + B \qquad \Gamma, x\colon A \vdash p\colon C \qquad \Gamma, y\colon B \vdash q\colon C}{\Gamma \vdash \mathsf{case}\, b\, \mathsf{of}\, \mathsf{inl}\, x \mapsto p;\ \mathsf{inr}\, y \mapsto q\colon C}$$

With this extension we can make do without *Bool* and *if-then-else* by encoding Bool as $1+1$, and by encoding *if-then-else* as follows:

$$\mathsf{if}\, b\, \mathsf{then}\, p\, \mathsf{else}\, q = \mathsf{case}\, b\, \mathsf{of}\, \mathsf{inl}\, x \mapsto p;\ \mathsf{inr}\, x \mapsto q.$$

In Haskell, coproducts are implemented as an algebraic data type:

```haskell
data  Either a b  =  Left a | Right b
```

Moreover, we can encode the "maybe" type constructor sending a type X to $MX = X + 1$ and introduce the following syntax:

$$(\mathbf{ret})\quad \frac{\Gamma \vdash t\colon A}{\Gamma \vdash \mathsf{return}\, t\colon MA} \qquad\qquad (\mathbf{bind})\quad \frac{\Gamma \vdash p\colon MB \qquad \Gamma, x\colon B \vdash q\colon MC}{\Gamma \vdash \mathsf{do}\, x \leftarrow p;\ q\colon MC}$$

where

$$\begin{aligned} \mathsf{return}\, t &= \mathsf{inl}\, t, \\ \mathsf{do}\, x \leftarrow p;\ q &= \mathsf{case}\, p\, \mathsf{of}\, \mathsf{inl}\, x \mapsto q;\ \mathsf{inl}\, y \mapsto \mathsf{inr}\, y. \end{aligned}$$

The intuition here: a term in context $\Gamma \vdash t\colon MX$ with $\Gamma = (x_1\colon A_1, \dots, x_n\colon A_n)$ models a partial function $(x_1, \dots, x_n) \mapsto t$, either sending a tuple $(x_1, \dots, x_n)$ to a value in X or to undefinedness; return and do correspondingly provide means for converting total functions to partial function and for composing partial functions correspondingly. This yields the second simplest example of a *monad*. This first one is the *identity monad* with $MX = X$, and with return and do defined in the obvious way.

In Haskell the maybe-monad is implemented as follows:

```
data Maybe a = Just a | Nothing

instance  Monad Maybe  where
    (Just x) >>= k      = k x
    Nothing  >>= _      = Nothing
```

In Haskell there is a very special monad called `IO` for interacting with the environment (reading/writing on the console, accessing/modifying the file system, the web, peripheral devises, sensors, etc). Thanks to the monadic abstraction, interaction with the environment is organized in the same style as with all other monads, e.g.

```
main :: IO ()
main = do
    putStrLn "Enter two lines"
    line1 <- getLine                            -- line1 :: String
    line2 <- getLine                            -- line2 :: String
    putStrLn ("you said: " ++ line1 ++ " and " ++ line2)
```

Remarkable and often confusing is the laziness of IO, which in conjunction with other lazy aspects can produce unexpected effects. Consider for instance

```
interact $ unlines . map reverse . lines
```

which reads a string from the console and prints it reversed in a loop. An analogous program

```
interact $ unlines . map (\s -> "your string is: " ++ !s) . lines
```

does not function as expected (starts printing before input is finished) because the input does not affect the initial part of the output, in contrast to the reverse-example where we needed to know the very last character of the input string to be able to say what the reversed string is. A possible fix is

```
interact $ unlines . map (\s -> seq (last s) "your string is " ++ s) . lines
```

Here `seq` is a very special build-in primitive, smuggling non-lazy semantics in. Very roughly `seq x y` runs `y` unless `x` unproductively diverges. Thus, e.g. strict function application is defined as

```
($!) :: (a -> b) -> a -> b
f $! x = x `seq` f x
```

An important property that is broken exclusively by `seq` is that f is no longer contextually equivalent to $\lambda x.\, f x$, which is otherwise true w.r.t. the call-by-name semantics (!) Indeed, `omega` and `\x -> omega x` can be distinguished by the context `const 0 $!` (but not with `const 0 $` !).

1.4 Denotational Semantics of PCF

Operational semantics is non-compositional, in the sense that it does not yield a function $[\![-]\!]$ from terms to meanings, so that for every n-ary term construct op, $[\![op(t_1, \ldots, t_n)]\!]$ could be calculated as a function of $[\![t_1]\!], \ldots, [\![t_n]\!]$. In particular, operational semantics does not directly define meanings of functions, hence we cannot express $[\![f\,t]\!]$ via $[\![f]\!]$ and $[\![t]\!]$.

In designing a denotational semantics (overall, but in our concrete case, for PCF) one would want to satisfy the following yardsticks:

- *Soundness:* if $p \Downarrow v$ then $[\![p]\!] = [\![v]\!]$;
- *Adequacy:* if $[\![p]\!] = [\![v]\!]$ with v being a ground value then $p \Downarrow v$;
- *Compositionality:* $[\![C[t]]\!] = [\![C]\!][\![t]\!]$ where we assume that t is closed and $[\![C]\!]$ is the same as $\lambda x.\, C[x]$.

Soundness and adequacy ensure that the denotational semantics sufficiently mimics the operational semantics. The soundness property is the most basic one and is usually easy to verify by induction over a derivation $p \Downarrow v$. Adequacy is usually considerably harder. Compositionality is where denotational semantics shines. For example, the proof of $f =_{\mathsf{ctx}} \lambda x.\, f x$ can be obtained as follows:

$$\begin{aligned}
C[f] \Downarrow v &\implies [\![C[f]]\!] = [\![v]\!] && /\!/ \text{ soundness}\\
&\iff [\![C]\!][\![f]\!] = [\![v]\!] && /\!/ \text{ compositionality}\\
&\iff [\![C]\!][\![\lambda x.\, f x]\!] = [\![v]\!] &&\\
&\iff [\![C[\lambda x.\, f x]]\!] = [\![v]\!] && /\!/ \text{ compositionality}\\
&\implies C[\lambda x.\, f x] \Downarrow v. && /\!/ \text{ adequacy}
\end{aligned}$$

This relies on the fact that $[\![f]\!] = [\![\lambda x.\, f x]\!]$, which will be an easy consequence of the definition of $[\![-]\!]$.

How can we define $[\![A]\!]$? It cannot be just a set of values of type A, e.g. $[\![Bool]\!] = \{\mathsf{True}, \mathsf{False}\}$. At least, $[\![Bool]\!]$ must include the divergence $\bot$. Is it enough to say that $[\![A]\!]$ collects the values of type A plus the divergence? No, for e.g. the function in $[\![A \to B]\!]$ must capture not only all total functions (the "values"), the totally undefined function (the "divergence"), but also all the partially defined functions in between (so, more or less defined). This issue propagates along type constructors, which is the reason we cannot think of $[\![A]\!]$ merely as a certain set, and $[\![f\colon A \to B]\!]$ merely as a certain function between sets $[\![A]\!]$ and $[\![B]\!]$. However, we can use as $[\![A]\!]$ *(ω-)complete partial orders (cpos)*, and as $[\![f\colon A \to B]\!]$ (ω-)continuous maps, between $[\![A]\!]$ and $[\![B]\!]$. This is a big idea of denotational semantics, proposed by Dana Scott.

Definition (Partial Orders)**.** A partial order $(A, \sqsubseteq)$ is a relation satisfying the following axioms:

- $a \sqsubseteq a$;
- $a \sqsubseteq b \wedge b \sqsubseteq c \Rightarrow a \sqsubseteq c$;
- $a \sqsubseteq b \wedge b \sqsubseteq a \Rightarrow a = b$.

Definition (Complete Partial Orders)**.** A(n ω-)cpo is a partial order $(A, \sqsubseteq)$, such that for any infinite chain

$$a_1 \sqsubseteq a_2 \sqsubseteq \ldots,$$

there is an a, such that

1. $\forall i.\, a_i \sqsubseteq a$;
2. $\forall i.\, a_i \sqsubseteq b \Rightarrow a \sqsubseteq b$.

We denote such a by $\bigsqcup_i a_i$. More, generally we write $\bigsqcup_{i \in I} a_i$ for any least upper bound (not necessarily of a chain) if $\forall i.\, a_i \sqsubseteq \bigsqcup_{i \in I} a_i$ and $\bigsqcup_{i \in I} a_i \sqsubseteq b$ once $\forall i.\, a_i \sqsubseteq b$.

Definition (Pointed Cpos)**.** A cpo $(A, \sqsubseteq)$ is pointed if it contains such an element $\bot$, that $\forall a \in A.\, \bot \sqsubseteq a$

Every set A is trivially a cpo $(A, \sqsubseteq)$ with $a \sqsubseteq b$ iff $a = b$.

Definition (Monotonicity, Continuity, Strictness)**.** A function $f\colon A \to B$ between partial orders is *monotone* if $a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$; a monotone function $f\colon A \to B$ between cpos $(A, \sqsubseteq)$ and $(B, \sqsubseteq)$ is *(Scott-)continuous* if for any chain $a_1 \sqsubseteq a_2 \sqsubseteq \ldots$:

$$f\Big(\bigsqcup\nolimits_i a_i\Big) = \bigsqcup\nolimits_i f(a_i)$$

A function $f\colon A \to B$ is *strict* if $f(\bot) = \bot$. This extends to the multi-ary functions in the obvious way, e.g. if-then-else is strict in the first argument, but not in the second and the third.

Definition ((Pre-)Domain)**.** We agree to refer to cpos as *pre-domains*, and to pointed cpos as *domains*.

1.4.1 Constructions on Predomains

Product of Predomains $A \times B = \{(a, b) \mid a \in A, b \in B\}$

$$(a_1, b_1) \sqsubseteq (a_2, b_2) \quad \text{if} \quad a_1 \sqsubseteq a_2 \quad \text{and} \quad b_1 \sqsubseteq b_2$$

Properties:

- Continuity of pairing: $\bigsqcup_i (a_i, b_i) = \big(\bigsqcup_i a_i, \bigsqcup_j b_j\big)$;
- Continuity of projections: $\mathsf{fst}\colon A \times B \to A$ and $\mathsf{snd}\colon A \times B \to B$ are continuous, i.e.: $\mathsf{fst}\big(\bigsqcup_j a_j\big) = \bigsqcup_j \mathsf{fst}\, a_j$, $\mathsf{snd}\big(\bigsqcup_j a_j\big) = \bigsqcup_j \mathsf{snd}\, a_j$;
- Products of domains are again domains with $(\bot, \bot)$ as the least element.

Lifting Predomains and Functions The correspondence $A \mapsto A_\bot$ defines a *lifing* of A where $A_\bot = \{\bot\} \uplus A = \{(0, \bot)\} \cup \{(1, a) \mid a \in A\}$.

$$a \sqsubseteq b \quad \text{if} \quad a = (0, \bot) \quad \text{or} \quad a = (1, a'), b = (1, b') \text{ and } a' \sqsubseteq b'$$

Let for any $a \in A$: $\lfloor a \rfloor = (1, a) \in A_\bot$. Since $(0, \bot)$ is now the bottom element of $A_\bot$, for notational simplicity we refer to it as $\bot$ again. Attempts to regard $\bot = (0, \bot)$ as a sort of recursive equation make no sense – $\bot$ on the left and $\bot$ on the right are distinguished by their context.

Let B be a domain and let $f \colon A \to B$ be continuous. Then we define $f^\star \colon A_\bot \to B$ as follows:

$$f^\star(x) = \begin{cases} f(y) & \text{if } x = \lfloor y \rfloor \\ \bot & \text{if } x = \bot \end{cases}$$

The result $f^\star$ is the *lifting* of f.

Notation. We use the point-full notation $(\mathsf{let}\, x = p \,\mathsf{in}\, q)$ alongside with the point-free one $(\lambda x.\, q)^\star(p)$ where $\lambda x.\, q \colon A \to B$ and $p \colon A_\bot$.

Properties:

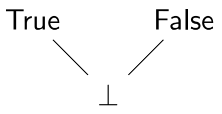
- $\lfloor - \rfloor$ is continuous: $\lfloor \bigsqcup_i a_i \rfloor = \bigsqcup_i \lfloor a_i \rfloor$.
- Lifting is continuous: $\left(\bigsqcup_i f_i\right)^\star = \bigsqcup_i f_i^\star$ where continuous functions are compared *pointwise*, that is $f \sqsubseteq g$ if $f(x) \sqsubseteq g(x)$ for any x (see the definition of function spaces bellow).

For every $op \colon X \times Y \to Z$ with X, Y, Z being sets, we define the *strict extension*:

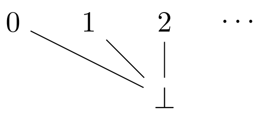
$$\begin{aligned} op_\bot &: X_\bot \times Y_\bot \to Z_\bot \\ op_\bot(p, q) &= \mathsf{let}\, x = p \,\mathsf{in}\, \mathsf{let}\, y = q \,\mathsf{in}\, \lfloor op(x, y) \rfloor \end{aligned}$$

Example (Flat Domains)**.** Given a set A, $A_\bot$ is called the *flat domain* over A, regarded as a trivially ordered set (i.e. $\sqsubseteq$ is $=$).

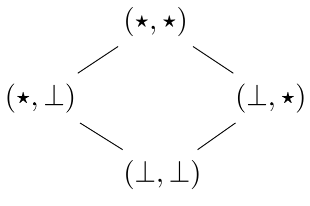
Bool$_\bot$:



Nat$_\bot$:



Non-example $1_\bot \times 1_\bot$:



Function Spaces Let $(A, \sqsubseteq)$ and $(B, \sqsubseteq)$ be two predomains. Then $(A \to_c B, \sqsubseteq)$ is the function space predomain, where

$$A \to_c B = \{f \colon A \to B \mid f \text{ is continuous}\}$$

and

$$f \sqsubseteq g \Leftrightarrow \forall x.\, f(x) \sqsubseteq g(x) \qquad \text{(pointwise)}$$

We define two operations:

$$\begin{aligned}&\mathsf{curry} \colon (A \times B \to_c C) \to (A \to_c (B \to_c C))\\ &(\mathsf{curry}\, f)(x)(y) = f(x, y)\end{aligned}$$

$$\begin{aligned}&\mathsf{uncurry} \colon (A \to_c (B \to_c C)) \to (A \times B \to_c C)\\ &(\mathsf{uncurry}\, f)(x, y) = f(x)(y)\end{aligned}$$

from which we can derive

$$\mathsf{ev} = \mathsf{uncurry}(\mathsf{id} \colon (A \to_c B) \to_c (A \to_c B)) \colon (A \to_c B) \times A \to_c B$$

Properties:

- curry and $\mathsf{uncurry}$ are continuous.
- If B is a domain then so is $A \to_c B$ with the bottom element being the completely undefined function $\lambda x.\, \bot$.

Theorem 1 (Kleene's Fixpoint Theorem)**.** Let f be a continuous function $f \colon D \to D$ over a domain D. Then

1. There is $\mu f \in D$ – the *least fixpoint of* f, i.e.
   a) $f(\mu f) = \mu f$
   b) $\forall x \in D.\, f(x) = x \Rightarrow \mu f \sqsubseteq x$
2. $\mu f = \bigsqcup_i f^i(\bot)$, where $f^0(x) = \bot$, $f^{i+1}(x) = f(f^i(x))$
3. $\mu f \in D$ is moreover the *least pre-fixpoint of* f, i.e.
   a) $f(\mu f) \sqsubseteq \mu f$
   b) $\forall x \in D.\, f(x) \sqsubseteq x \Rightarrow \mu f \sqsubseteq x$

Proof. Let us first show that μf as defined in clause 2 is a fixpoint of f. Indeed, $f(\mu f) = f\Big(\bigsqcup_i f^i(\bot)\Big) = \Big(\bigsqcup_i f^{i+1}(\bot)\Big) = \mu f$. Hence is it also a prefixpoint. Let us show that it is the least one. Suppose that $x \in D$ is another prefixpoint, i.e. $f(x) \sqsubseteq x$. From $\bot \sqsubseteq x$, inductively, $f^i(\bot) \sqsubseteq f^i(x) = x$, hence $\mu f = \bigsqcup_i f^i(\bot) \sqsubseteq x$. Since μf is the least prefixpoint and a fixpoint, it is in particular the least fixpoint. □

Example. Consider $f_i \colon \mathbb{N} \to \mathbb{N}_\bot$ with $i \in \mathbb{N}$:

$$f_0(n) = \bot \qquad (n \geqslant 0)$$

$$f_1(0) = \lfloor 1 \rfloor, f_1(n) = \bot \qquad (n \geqslant 1)$$

$$
\begin{aligned}
& f_2(0) = \lfloor 1 \rfloor, f_2(1) = \lfloor 1 \rfloor, f_2(n) = \bot && (n \geqslant 2)\\
& f_3(0) = \lfloor 1 \rfloor, f_3(1) = \lfloor 1 \rfloor, f_3(2) = \lfloor 2 \rfloor, f_3(n) = \bot && (x \geqslant 3)\\
& f_4(0) = \lfloor 1 \rfloor, f_4(1) = \lfloor 1 \rfloor, f_4(2) = \lfloor 2 \rfloor, f_4(3) = \lfloor 6 \rfloor, f_4(n) = \bot && (x \geqslant 4)\\
& \vdots
\end{aligned}
$$

It's easy to see that $f_i \sqsubseteq f_{i+1}$ for any $i \in \mathbb{N}$. Let

$$f = \bigsqcup_i f_i$$

Again, it is easy to see that for every $f(n) = n!$. By Kleene fixpoint theorem, we can interpret this as the fact that f is a solutions of a recursive equation, defining the factorial function. Note that

$$f_{i+1} = F(f_i) \qquad \forall (i \in \mathbb{N})$$

where $F\colon (\mathbb{N} \to \mathbb{N}_\bot) \to_c (\mathbb{N} \to \mathbb{N}_\bot)$ is as follows:

$$F(g\colon \mathbb{N} \to \mathbb{N}_\bot)(n \in \mathbb{N}) = \begin{cases} \lfloor 1 \rfloor & \text{if } n = 0, 1 \\ \lfloor n \rfloor \cdot_\bot g(n-1) & \text{if } n > 1 \end{cases}$$

which is the defining expression for the factorial. So, by Kleene fixpoint theorem:

$$\mu F = \bigsqcup_i F^i(\bot) = \bigsqcup_i f_i = f.$$

We can reformulate this result as follows: $f\colon \mathbb{N} \to \mathbb{N}_\bot$ is a solution of the following system of equations:

$$
\begin{aligned}
f(0) &= \lfloor 1 \rfloor \\
f(1) &= \lfloor 1 \rfloor \\
f(n) &= \mathsf{let}\, m \coloneqq f(n-1)\, \mathsf{in} \lfloor n \cdot m \rfloor \qquad (n > 1)
\end{aligned}
$$

and it is the least such solution (!)

Proposition. $\mu\colon (D \to_c D) \to D$ is continuous.

Definition. Given a domain X, let $\mathsf{ifThenElse}\colon \mathbb{B}_\bot \times X \times X \to X$:

$$\mathsf{ifThenElse}(b, x, y) = \begin{cases} x & \text{if } b = \lfloor \mathsf{True} \rfloor \\ y & \text{if } b = \lfloor \mathsf{False} \rfloor \\ \bot & \text{otherwise} \end{cases}$$

Proposition. $\mathsf{ifThenElse}$ is continuous.

1.4.2 CBN Denotational Semantics

We assign to every type A a *domain* $[\![A]\!]$ as follows:

- $[\![1]\!] = \{\star\}$;
- $[\![Nat]\!] = \mathbb{N}_\bot$ (flat domain of naturals);
- $[\![Bool]\!] = \mathbb{B}_\bot$ (flat domain of Booleans);
- $[\![A \times B]\!] = [\![A]\!] \times [\![B]\!]$;
- $[\![A \to B]\!] = [\![A]\!] \to_c [\![B]\!]$.

Now, given a term in context $\Gamma \vdash t\colon A$ where $\Gamma = x_1\colon A_1, \ldots, x_n\colon A_n$ the semantics $[\![\Gamma \vdash t\colon A]\!]$ is a continuous function $[\![A_1]\!] \times \ldots \times [\![A_n]\!] \to [\![A]\!]$ recursively computed according to the following clauses where $[\![\cdots]\!]_\rho$ reads as $[\![\cdots]\!](\rho)$:

- $[\![\Gamma \vdash x_i\colon A_i]\!]_\rho = \mathsf{pr}_i(\rho)$ where $\mathsf{pr}_i\colon [\![A_1]\!] \times \ldots \times [\![A_n]\!] \to [\![A_i]\!]$ is the i-th projection;
- $[\![\Gamma \vdash \star\colon 1]\!]_\rho = \star$;
- $[\![\Gamma \vdash b\colon Bool]\!]_\rho = \lfloor b \rfloor$;
- $[\![\Gamma \vdash n\colon Nat]\!]_\rho = \lfloor n \rfloor$;
- $[\![\Gamma \vdash f(t,s)\colon A]\!]_\rho = f_\bot([\![\Gamma \vdash t\colon B]\!]_\rho, [\![\Gamma \vdash s\colon C]\!]_\rho) \qquad (f \in \{\wedge, \to, +, -, \times, =\})$;
- $[\![\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t\colon A]\!]_\rho = \mathsf{ifThenElse}([\![\Gamma \vdash b\colon Bool]\!]_\rho, [\![\Gamma \vdash s\colon A]\!]_\rho, [\![\Gamma \vdash t\colon A]\!]_\rho)$;
- $[\![\Gamma \vdash \langle t, s\rangle\colon A \times B]\!]_\rho = \langle [\![\Gamma \vdash t\colon A]\!]_\rho, [\![\Gamma \vdash s\colon B]\!]_\rho \rangle$;
- $[\![\Gamma \vdash \mathsf{fst}\, t\colon A]\!]_\rho = \mathsf{fst}[\![\Gamma \vdash t\colon A \times B]\!]_\rho$;
- $[\![\Gamma \vdash \mathsf{snd}\, t\colon B]\!]_\rho = \mathsf{snd}[\![\Gamma \vdash t\colon A \times B]\!]_\rho$;
- $[\![\Gamma \vdash \lambda x.\, t\colon A \to B]\!]_\rho = (\mathsf{curry}\, [\![\Gamma, x\colon A \vdash t\colon B]\!])(\rho)$;
- $[\![\Gamma \vdash s\, t\colon B]\!]_\rho = \mathsf{ev}([\![\Gamma \vdash s\colon A \to B]\!]_\rho, [\![\Gamma \vdash t\colon A]\!]_\rho)$;
- $[\![\Gamma \vdash Y_A\colon (A \to A) \to A]\!]_\rho = \mu$.

Lemma (Substitution Lemma)**.** Given $\Gamma \vdash q\colon A$, $\Gamma, x\colon A \vdash p\colon B$ and $\rho \in [\![\Gamma]\!]$

$$[\![\Gamma \vdash p[q/x]\colon B]\!]_\rho = [\![\Gamma, x\colon A \vdash p\colon B]\!](\rho, [\![\Gamma \vdash q\colon A]\!]_\rho)$$

Proof. Induction over the structure of p. Let us consider the there last clauses in the semantics for p, which are the only non-trivial ones.

- $p = \lambda y.\, t$ with some $\Gamma, y\colon C \vdash t\colon D$ and then $B = C \to D$. It follows by assumption that $x \neq y$. Then, by induction,

$$\begin{aligned}
[\![\Gamma \vdash p[q/x]\colon B]\!]_\rho &= [\![\Gamma \vdash \lambda y.\, t[q/x]\colon B]\!]_\rho \\
&= (\mathsf{curry}[\![\Gamma, y\colon C \vdash t[q/x]\colon D]\!])(\rho) \\
&= (\mathsf{curry}([\![\Gamma, y\colon C, x\colon A \vdash t\colon D]\!] \circ (\mathsf{id}, [\![\Gamma, y\colon C \vdash q\colon A]\!])))(\rho) \\
&= (\mathsf{curry}[\![\Gamma, x\colon A, y\colon C \vdash t\colon D]\!])(\rho, [\![\Gamma \vdash q\colon A]\!]_\rho) \\
&= [\![\Gamma, x\colon A \vdash \lambda y.\, t\colon B]\!](\rho, [\![\Gamma \vdash q\colon A]\!]_\rho) \\
&= [\![\Gamma, x\colon A \vdash p\colon B]\!](\rho, [\![\Gamma \vdash q\colon A]\!]_\rho).
\end{aligned}$$

- $p = s\,t$ with some $\Gamma, x\colon A \vdash t\colon C$ and $\Gamma, x\colon A \vdash s\colon C \to B$. Then, by induction,

$$\begin{aligned}
[\![\Gamma \vdash p[q/x]\colon B]\!]_\rho &= [\![\Gamma \vdash (s[q/x])\,(t[q/x])\colon B]\!]_\rho \\
&= [\![\Gamma \vdash s[q/x]\colon C \to B]\!]_\rho([\![\Gamma \vdash t[q/x]\colon C]\!]_\rho) \\
&= ([\![\Gamma, x\colon A \vdash s\colon C \to B]\!](\rho, [\![\Gamma \vdash q\colon A]\!]_\rho)) \\
&\qquad ([\![\Gamma, x\colon A \vdash t\colon C]\!](\rho, [\![\Gamma \vdash q\colon A]\!]_\rho)) \\
&= [\![\Gamma, x\colon A \vdash s\,t\colon B]\!](\rho, [\![\Gamma \vdash q\colon A]\!]_\rho) \\
&= [\![\Gamma, x\colon A \vdash p\colon B]\!](\rho, [\![\Gamma \vdash q\colon A]\!]_\rho).
\end{aligned}$$

- $p = Y_B\, f$ with some $\Gamma, x\colon A \vdash f\colon B \to B$. Analogously to the previous clauses:

$$\begin{aligned}
[\![\Gamma \vdash p[q/x]\colon B]\!]_\rho &= [\![\Gamma \vdash (Y_B\, f)[q/x]\colon B]\!]_\rho \\
&= [\![\Gamma \vdash Y_B\, f[q/x]\colon B]\!]_\rho \\
&= \mu[\![\Gamma \vdash f[q/x]\colon B \to B]\!]_\rho \\
&= \mu([\![\Gamma, x\colon A \vdash f\colon B \to B]\!](\rho, [\![\Gamma \vdash q\colon A]\!]_\rho)) \\
&= [\![\Gamma, x\colon A \vdash \mu f\colon B]\!](\rho, [\![\Gamma \vdash q\colon A]\!]_\rho) \\
&= [\![\Gamma, x\colon A \vdash p\colon B]\!](\rho, [\![\Gamma \vdash q\colon A]\!]_\rho).
\end{aligned}$$

□

Definition (Soundness)**.** A denotational semantics is *sound* if

$$p \Downarrow v \Rightarrow [\![p]\!] = v$$

Definition (Adequacy)**.** A denotational semantics is *adequate*, if

$$[\![p]\!] = v \Rightarrow p \Downarrow v \quad \text{if the type of } p \text{ is either 1 or } \mathit{Bool} \text{ or } \mathit{Nat}$$

for every value v.

Proposition. The presented call-by-name denotational semantics is sound and adequate with respect to $\Downarrow_{\mathrm{cbn}}$.

Soundness is typically easy to prove: using the equivalence of big-step and small-step semantics, it suffices to prove that $p \to q$ entails $[\![p]\!] = [\![q]\!]$; then $p \Downarrow v$ entails $p \to^\star v$ and we are done by induction over the length of this reduction. Proving adequacy is usually much harder and requires new methods.

Assuming soundness, we can equivalently reformulate adequacy as follows: by contraposition, it says that for all values v, $\neg(p \Downarrow v) \Rightarrow [\![p]\!] \neq v$, in particular, if $p \Uparrow$ then $[\![p]\!] \neq v$ for any value v, i.e. $[\![p]\!] = \bot$. This is the only instance that does not follow from soundness. Indeed, if $\neg(p \Downarrow v)$, but $p \Downarrow v'$ with $v' \neq v$ then by soundness $[\![p]\!] = v'$ and hence $[\![p]\!] \neq v$. In summary, we equivalently switched to the implication:

$$p \Uparrow \Rightarrow [\![p]\!] = \bot.$$

Recall the following property of contextual equivalence.

Proposition. Let $\Gamma \vdash s\colon A$ and $\Gamma \vdash t\colon A$ be two programs in context, A be a ground type and let Γ be empty. Then $s =_{\mathsf{ctx}} t$ iff for every value v, $s \Downarrow v$ iff $t \Downarrow v$.

We now can prove it. The left to right direction is clear. Assume the right hand side and prove $s =_{\mathsf{ctx}} t$. If $s \Downarrow v$ for some v then $t \Downarrow v$ and by soundness, $\llbracket s \rrbracket = v = \llbracket t \rrbracket$. If $s \Uparrow$ then $t \Uparrow$, and by adequacy, $\llbracket s \rrbracket = \bot = \llbracket t \rrbracket$. In any case, we have $\llbracket C[s] \rrbracket = \llbracket C[t] \rrbracket$ with $C = \Box$. The general case follows by induction on C. Then $C[s] \Downarrow v$ by adequacy entails $C[t] \Downarrow v$ and vice versa.

1.4.3 Failure of Full Abstraction

Note the implication

$$\llbracket p \rrbracket = \llbracket q \rrbracket \Rightarrow p =_{\mathsf{ctx}} q$$

where p and q are closed programs of the same type. Indeed, for very suitable context C of ground output type,

$$\begin{aligned}
C[p] \Downarrow v &\iff \llbracket C[p] \rrbracket = \lfloor v \rfloor && \text{// sound. and adeq.}\\
&\iff \llbracket C \rrbracket[\llbracket p \rrbracket] = \lfloor v \rfloor && \text{// compositionality}\\
&\iff \llbracket C \rrbracket[\llbracket q \rrbracket] = \lfloor v \rfloor && \text{// assumption}\\
&\iff \llbracket C[q] \rrbracket = \lfloor v \rfloor && \text{// compositionality}\\
&\iff C[q] \Downarrow v && \text{// sound. and adeq.}
\end{aligned}$$

We have thus obtained a fundamental relation between operational and denotational semantics: contextually equivalent programs are necessarily denotationally equal. The opposite implication

$$p =_{\mathsf{ctx}} q \Rightarrow \llbracket p \rrbracket = \llbracket q \rrbracket$$

is called *full abstraction*, and it would provide the highest degree of satisfaction, for it would mean that operational sematnics and denotational semantics agree (as far as program equivalence is concerned). However, for our precent semantics full abstraction fails (!), and the reason for it is instructive.

Consider the following PCF-function in Haskell syntax:

```haskell
t :: Bool -> (Bool -> Bool -> Bool) -> Bool
t b f = if (f True omega).
        then if (f omega True).
             then if (f False False) then omega else b
             else omega
        else omega
          where omega = omega
```

It can be shown that `(t True)` and `(t False)` are contextually equivalent, however, they are not denotationally equivalent. The reason for it is that we cannot test `(t True)` and `(t False)` on the parallel-or function, which is described as follows: $\mathit{por}(\mathsf{True}, x) = \mathit{por}(x, \mathsf{True}) = \mathsf{True}$, $\mathit{por}(\mathsf{False}, \mathsf{False}) = \mathsf{False}$ and $\mathit{por}(x, y) = \bot$ otherwise. This function is not definable in PCF, but it is a continuous function, and thus, it can be used as a witness that `(t True)` and `(t False)` are denotationally distinct.

1.4.4 CBV Denotational Semantics

We we assign to every type A a predomain $[\![A]\!]$ as follows:

- $[\![1]\!] = \{\star\}$;
- $[\![Nat]\!] = \mathbb{N}$;
- $[\![Bool]\!] = \mathbb{B}$;
- $[\![A \times B]\!] = [\![A]\!] \times [\![B]\!]$;
- $[\![A \to B]\!] = [\![A]\!] \to [\![B]\!]_\bot$.

Now, the semantics of a term in context $\Gamma \vdash t\colon A$ with $\Gamma = (x_1\colon A_1, \ldots, x_n\colon A_n)$ is a continuous function $[\![A_1]\!] \times \ldots \times [\![A_n]\!] \to [\![A]\!]_\bot$ defined by structural induction as follows.

- $[\![\Gamma \vdash x_i\colon A_i]\!]_\rho = \lfloor \mathsf{pr}_i(\rho) \rfloor$;
- $[\![\Gamma \vdash n\colon Nat]\!]_\rho = \lfloor n \rfloor$;
- $[\![\Gamma \vdash b\colon Bool]\!]_\rho = \lfloor b \rfloor$;
- $[\![\Gamma \vdash f(t,s)\colon A]\!]_\rho = f_\bot([\![\Gamma \vdash t\colon B]\!]_\rho, [\![\Gamma \vdash s\colon C]\!]_\rho) \qquad (f \in \{\wedge, \to, +, -, \times, =\})$;
- $[\![\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t\colon A]\!]_\rho = \mathsf{ifThenElse}([\![\Gamma \vdash b\colon Bool]\!]_\rho, [\![\Gamma \vdash s\colon A]\!]_\rho, [\![\Gamma \vdash t\colon A]\!]_\rho)$;
- $[\![\Gamma \vdash \langle t,s\rangle\colon A \times B]\!]_\rho = \mathsf{let}\, x \coloneqq [\![\Gamma \vdash t\colon A]\!]_\rho\, \mathsf{in}\, \mathsf{let}\, y \coloneqq [\![\Gamma \vdash s\colon B]\!]_\rho\, \mathsf{in} \lfloor \langle x,y\rangle \rfloor$;
- $[\![\Gamma \vdash \mathsf{fst}\, t\colon A]\!]_\rho = \mathsf{let}\, v \coloneqq [\![\Gamma \vdash t\colon A \times B]\!]_\rho\, \mathsf{in} \lfloor \mathsf{fst}\, v \rfloor$;
- $[\![\Gamma \vdash \mathsf{snd}\, t\colon B]\!]_\rho = \mathsf{let}\, v \coloneqq [\![\Gamma \vdash t\colon A \times B]\!]_\rho\, \mathsf{in} \lfloor \mathsf{snd}\, v \rfloor$;
- $[\![\Gamma \vdash \lambda x.\, t\colon A \to B]\!]_\rho = \lfloor (\mathsf{curry}\, [\![\Gamma, x\colon A \vdash t\colon B]\!])(\rho) \rfloor$;
- $[\![\Gamma \vdash s\, t\colon B]\!]_\rho = \mathsf{let}\, v \coloneqq [\![\Gamma \vdash t\colon A]\!]_\rho\, \mathsf{in}\, \mathsf{let}\, f \coloneqq [\![\Gamma \vdash s\colon A \to B]\!]_\rho\, \mathsf{in}\, \mathsf{ev}(f, v)$;
- $[\![\Gamma \vdash Y_{A \to B}]\!]_\rho = \lambda f.\, \mu(\lambda g.\, f(\lambda x.\ \mathsf{let}\, h \coloneqq g\, \mathsf{in}\, h(x)))$ where $f\colon ([\![A]\!] \to [\![B]\!]_\bot) \to ([\![A]\!] \to [\![B]\!]_\bot)_\bot$ and $g \in ([\![A]\!] \to [\![B]\!]_\bot)_\bot$.

The analogue of the substitution lemma is as follows.

Lemma (Substitution Lemma)**.** Given $\Gamma \vdash q\colon A$, $\Gamma, x\colon A \vdash p\colon B$ and $\rho \in [\![\Gamma]\!]$,

$$[\![\Gamma \vdash p[q/x]\colon B]\!]_\rho = \mathsf{let}\, v \coloneqq [\![\Gamma \vdash q\colon A]\!]_\rho\, \mathsf{in} [\![\Gamma, x\colon A \vdash p\colon B]\!](\rho, v)$$

provided that q is of the form $\lambda z.\, r$.

In contrast to the call-by-name case, the assumption that $q = \lambda z.\, r$ is essential. For example, if q diverges, but p does not depend on x, we would have $[\![\Gamma \vdash p\colon B]\!]$ on the left-hand side and $\bot$ on the right-hand side.

Proof. The proof is by structural induction over p. Again, only the last three clauses in the definition of semantics of p are sophisticated. Still the other ones require some properties of the let-construct (commutativity and copyability).

Assume that $\Gamma, z\colon E \vdash r\colon F$, i.e. $A = E \to F$.

- $p = \lambda y.\, t$ with some $\Gamma, y\colon C, x\colon A \vdash t\colon D$ and then $B = C \to D$. It follows by assumption that $x \neq y$. Let us fix $c \in [\![C]\!]$, $\rho \in [\![\Gamma]\!]$ and let $s = \mathsf{let}\, v = [\![\Gamma \vdash q\colon A]\!]_\rho \,\mathsf{in}\, [\![\Gamma, x\colon A \vdash p\colon B]\!](\rho, v)$. It is easy to check that $s = \lfloor g \rfloor$ for some g. Then

$$\begin{aligned}
&[\![\Gamma, y\colon C \vdash t[q/x]\colon D]\!](\rho, c)\\
&\quad = \mathsf{let}\, v = [\![\Gamma, y\colon C \vdash q\colon A]\!](\rho, c) \,\mathsf{in}\, [\![\Gamma, y\colon C, x\colon A \vdash t\colon D]\!](\rho, c, v)\\
&\quad = \mathsf{let}\, v = [\![\Gamma \vdash q\colon A]\!]_\rho \,\mathsf{in}\, [\![\Gamma, x\colon A, y\colon C \vdash t\colon D]\!](\rho, v, c)\\
&\quad = \mathsf{let}\, v = [\![\Gamma \vdash q\colon A]\!]_\rho \,\mathsf{in}\, \mathsf{let}\, f = [\![\Gamma, x\colon A \vdash p\colon B]\!](\rho, v) \,\mathsf{in}\, f(c)\\
&\quad = \mathsf{let}\, f = (\mathsf{let}\, v = [\![\Gamma \vdash q\colon A]\!]_\rho \,\mathsf{in}\, [\![\Gamma, x\colon A \vdash p\colon B]\!](\rho, v)) \,\mathsf{in}\, f(c)\\
&\quad = \mathsf{let}\, f = \lfloor g \rfloor \,\mathsf{in}\, f(c)\\
&\quad = g(c)
\end{aligned}$$

using the fact that q does not depend on y. Now

$$\begin{aligned}
&[\![\Gamma \vdash p[q/x]\colon B]\!]_\rho\\
&\quad = [\![\Gamma \vdash \lambda y.\, t[q/x]\colon B]\!]_\rho\\
&\quad = \lfloor (\mathsf{curry}[\![\Gamma, y\colon C \vdash t[q/x]\colon D]\!])(\rho) \rfloor\\
&\quad = \lfloor g \rfloor\\
&\quad = s.
\end{aligned}$$

- $p = s\, t$ with some $\Gamma, x\colon A \vdash t\colon C$ and $\Gamma, x\colon A \vdash s\colon C \to B$. Then, by induction,

$$\begin{aligned}
[\![\Gamma \vdash p[q/x]\colon B]\!]_\rho &= [\![\Gamma \vdash (s[q/x])\,(t[q/x])\colon B]\!]_\rho\\
&= \mathsf{let}\, v = [\![\Gamma \vdash t[q/x]\colon C]\!]_\rho \;\mathsf{in}\\
&\qquad \mathsf{let}\, f = [\![\Gamma \vdash s[q/x]\colon C \to B]\!]_\rho \,\mathsf{in}\, f(v)\\
&= \mathsf{let}\, w = [\![\Gamma \vdash q\colon A]\!]_\rho \,\mathsf{in}\, \mathsf{let}\, v = [\![\Gamma, x\colon A \vdash t\colon C]\!](\rho, w) \;\mathsf{in}\\
&\qquad \mathsf{let}\, f = [\![\Gamma, x\colon A \vdash s\colon C \to B]\!](\rho, w) \,\mathsf{in}\, f(v)\\
&= \mathsf{let}\, w = [\![\Gamma \vdash q\colon A]\!]_\rho \,\mathsf{in}\, [\![\Gamma, x\colon A \vdash s\, t\colon B]\!](\rho, w).
\end{aligned}$$

- $p = Y_B\, f$ with some $\Gamma, x\colon A \vdash f\colon (C \to D) \to (C \to D)$, hence $B = (C \to D)$. Note that for a suitable w, $[\![\Gamma \vdash q\colon A]\!]_\rho = \lfloor w \rfloor$. Then

$$\begin{aligned}
[\![\Gamma \vdash p[q/x]\colon B]\!]_\rho &= [\![\Gamma \vdash (Y_B\, f)[q/x]\colon B]\!]_\rho\\
&= [\![\Gamma \vdash Y_B\, f[q/x]\colon B]\!]_\rho\\
&= \mu(g)\\
&= [\![\Gamma, x\colon A \vdash Y_B f\colon B]\!](\rho, w)\\
&= \mathsf{let}\, v = [\![\Gamma \vdash q\colon A]\!]_\rho \,\mathsf{in}\, [\![\Gamma, x\colon A \vdash p\colon B]\!](\rho, v).
\end{aligned}$$

where $g(p) = \mathsf{let}\, h = [\![\Gamma, x\colon A \vdash f\colon B \to B]\!](\rho, w) \,\mathsf{in}\, h(u(p))$ and $u(p)(x) = \mathsf{let}\, h = p \,\mathsf{in}\, h(x)$. □

Proposition. The CBV semantics of PCF is sound and adequate.

Proposition (let-unit-1)**.** $\mathsf{let}\, x = \lfloor t \rfloor \,\mathsf{in}\, p = p[t/x]$.

Proof.

$$\mathsf{let}\, x = \lfloor t \rfloor \,\mathsf{in}\, p = (\lambda x.\, p)^\star \lfloor t \rfloor \quad = \quad \begin{cases} (\lambda x.\, p)(s) & \text{if} \quad \lfloor t \rfloor = \lfloor s \rfloor \\ \bot & \text{otherwise} \end{cases} \quad = \quad \begin{cases} (\lambda x.\, p)t \\ p[t/x] \end{cases}$$

□

Proposition (let-unit-2)**.** $\mathsf{let}\, x = p \,\mathsf{in}\, \lfloor x \rfloor = p$.

Proof. $\mathsf{let}\, x = p \,\mathsf{in}\, \lfloor x \rfloor = (\lambda x.\, \lfloor x \rfloor)^\star(p) = (\lambda x.\, x)(p) = p$. □

Proposition (let-assoc)**.**

$$\mathsf{let}\, x = p \,\mathsf{in}(\mathsf{let}\, y = q \,\mathsf{in}\, r) = \mathsf{let}\, y = (\mathsf{let}\, x = p \,\mathsf{in}\, q) \,\mathsf{in}\, r.$$

where $x \notin \mathrm{Free}(r)$.

Alternatively, the three laws for the let-operator can be presented in the pointfree form as follows:

$$f^\star \eta = f \qquad\qquad \eta^\star = \mathsf{id} \qquad\qquad f^\star g^\star = (f^\star g)^\star$$

where $\eta\colon A \to A_\bot$ sends x to $\lfloor x \rfloor$. These are known as *monad laws*, and they identify the map $A \mapsto A_\bot$ as a monad whose *unit* is $\lfloor - \rfloor$ and whose *Kleisli lifting* is the operation $(-)^\star$.

Thus, a monad can be understood as a certain type constructor that transforms *values* to *computations* and induces a notion of generalized function, carrying a certain *(side-)effect* in contrast to “normal functions”. The side-effect of the lifting monad is *divergence*. Further side-effects that can be abstracted in monads include

- abortion,
- non-determinism,
- store,
- input/output,

and in fact many others. In order to make these considerations rigorous, we proceed with the basic concepts of category theory. As we will see, monads is a genuinely categorical concept.

2 Categories and Monads

In this chapter, we proceed to identify *categorical* structures that are relevant for semantics (and functional programming), specifically to that its fragment we have considered previously. Category theory provides a language to deal with structures in such a way that the language itself abstracts from a concrete way these structures are implemented. Instead, in category theory we describe structures of interest by their *universal properties*, i.e. such properties that define them uniquely *up-to isomorphism.* For example, in set theory the property of being a *singleton* does not define any particular set: both { 🍒 } and { 🍌 } are singleton sets, and many more. But all singletons are *isomorphic*, and in fact, as long as structures of interest are isomorphic, choosing a specific one is a matter of convention, so it makes sense to characterise singletons so as to remove any reference to a concrete implementation. In category theory, this is done via the notion of *terminal object.* This illustrates one of the key insights of category theory: one should work with universal properties of concepts, uniquely (up to isomorphism) characterising the concept, instead of concrete representations.

2.1 Introducing Monads

Let us consider the *do-notation*, as a generalization of our previous *let-notation.* The idea is to capture the most abstract properties of computation, e.g. the let-notation also satisfies the following *commutativity* property:

$$\mathsf{let}\, x = p \,\mathsf{in}\, \mathsf{let}\, y = q \,\mathsf{in} \lfloor \langle x, y \rangle \rfloor \;=\; \mathsf{let}\, y = q \,\mathsf{in}\, \mathsf{let}\, x = p \,\mathsf{in} \lfloor \langle x, y \rangle \rfloor,$$

which is not abstract enough: if p writes to a store and q reads from that store the order in which p and q are executed obviously matters.

Essentially we introduce two term constructs:

$$\mathsf{do}\, x \leftarrow \underbrace{\;p\;}_{TA} ;\ \underbrace{\;f\;}_{A \to TB}(x) \qquad\qquad\qquad \mathsf{ret}\colon A \to TA$$

In conjunction with other (obvious) term constructs this forms what is known as (first-order) *computational metalanguage* whose syntax is *Haskell's do-notation.*

Essentially, we need monads to interpret this notation. Monads is a categorical concept. We need few preliminaries to introduce them formally.

Definition (Category)**.** A Category $\mathcal{C}$ consists of a collection of objects $|\mathcal{C}|$ and a collection of morphisms $\mathcal{C}(A, B)$ (also written as $\mathsf{Hom}_{\mathcal{C}}(A, B)$, or $\mathsf{Hom}(A, B)$ if $\mathcal{C}$ is clear) for all $A, B \in |\mathcal{C}|$, such that the following properties hold:

- for every $A \in |\mathcal{C}|$ there is an *identity morphism* $\mathsf{id}_A \in \mathcal{C}(A, A)$;
- for any $f \in \mathcal{C}(B, C)$ and $g \in \mathcal{C}(A, B)$ we can form a *composition* $f \circ g \in \mathcal{C}(A, C)$;
- $\mathsf{id} \circ f = f$, $f \circ \mathsf{id} = f$, $(f \circ g) \circ h = f \circ (g \circ h)$.

We also write $f\colon A \to B$ instead of $f \in \mathcal{C}(A, B) = \mathsf{Hom}(A, B)$.

A "collection" in the definition of a category is in fact a *"class"*, i.e. something generally larger than a set, e.g. the "set of all sets" does not make sense, but "all sets" form a class. Categories in which any $\mathsf{Hom}(A, B)$ is a set are called *locally small* and the categories in which $|\mathcal{C}|$ is a set are called *small*. Most of our examples of categories are locally small but not small.

Example. Examples of categories:

- Sets: $|\text{Sets}|$ = "all sets" and $\mathsf{Hom}(A, B)$ = "functions from A to B".
- Cpo: $|\text{Cpo}|$ = "all cpos" and $\mathsf{Hom}(A, B)$ = "continuous functions from A to B".
- Rel: $|\text{Rel}|$ = "all sets" and $\mathsf{Hom}(A, B)$ = "relations $R \subseteq A \times B$" with

$$\begin{aligned} \mathsf{id}_A &= \{(x, x) \mid x \in A\} \\ R \circ S &= \{(x, z) \in A \times C \mid \exists y \in B.\, (x, y) \in R, (y, z) \in S\} \end{aligned}$$

- PFun: $|\text{PFun}|$ = "all sets" and $\mathsf{Hom}(A, B)$ = "partial functions from A to B".

Definition (Commutative Diagrams)**.** We consider diagrams whose nodes are labeled with objects and whose edges are oriented and labelled with morphisms. A diagram *commutes* if all paths with the same start and endpoint produce equal morphisms (the morphism are formed by composing the labels along paths).

For example, the axioms for identity can be stated as follows:

$$\begin{tikzcd}
A \arrow[r, "\mathsf{id}"] \arrow[dr, bend right, "f"'] & A \arrow[d, "f"] & A \arrow[l, "\mathsf{id}"'] \arrow[dl, bend left, "f"] \\
& B &
\end{tikzcd}$$

Curiously, we cannot express associativity of composition in this way, because it is already baked in to the diagrammatic language.

In category theory, it is customary to prove equations between morphisms $f = g$ *"by diagram chasing"*, that is, by producing a commutative diagram, from which a chain of equations $f = f' = f'' = \ldots = g' = g$ can be read out. Importantly, not every commutative diagram produces a proof like this. For example, the diagram

$$\begin{tikzcd}
\bullet \arrow[rr, "a"] \arrow[dd, "c"'] \arrow[dr] & & \bullet \arrow[dd, "b"] \\
& \bullet \arrow[ur] \arrow[dl] & \\
\bullet \arrow[rr, "d"] & & \bullet \arrow[ul]
\end{tikzcd}$$

does not prove the equation $ba = dc$ even though all the triangles commute.

2.1.1 Products and Coproducts

Definition (Binary Products)**.** A (binary) product of objects A, B in a category $\mathcal{C}$ is a triple $(A \times B \in |\mathcal{C}|, \mathsf{fst}\colon A \times B \to A, \mathsf{snd}\colon A \times B \to B)$, such that for any $C \in |\mathcal{C}|$ with $f\colon C \to A, g\colon C \to B$, there is unique (!) morphism $\langle f, g\rangle\colon C \to A \times B$, such that the following diagram commutes:

$$\begin{tikzcd}
& C \arrow[dl, bend right, "f"'] \arrow[dr, bend left, "g"] \arrow[d, "\langle f{,}s\rangle"'] & \\
A & A \times B \arrow[l, "\mathsf{fst}"] \arrow[r, "\mathsf{snd}"'] & B
\end{tikzcd}$$

As a text: $\mathsf{fst} \circ \langle f, g\rangle = f$, $\mathsf{snd} \circ \langle f, g\rangle = g$. The morphisms fst and snd are called *(left and right) projections* and the operation $f, g \mapsto \langle f, g\rangle$ is called *pairing.*

Example.

- In Sets, products are Cartesian products.
- In Cpo, products are products of Cpos.

Definition (Terminal Object)**.** A terminal object is an object $1 \in |\mathcal{C}|$, such that for any $A \in |\mathcal{C}|$, there is a unique morphism: $!_A\colon A \to 1$

Definition (Cartesian Category)**.** A *Cartesian category* is a category with a terminal object and binary products.

Equivalently, a Cartesian category is the one which has all finite products: products of a nonempty finite number of components are obviously induced by binary products, the product of the empty family of components is the terminal object.

Examples: Sets and functions, Cpos and continuous functions, . . .

Definition (Isomorphism)**.** An *isomorphism* between objects A and B in a category $\mathcal{C}$ is given by a pair of morphisms: $f\colon A \to B$, $g\colon B \to A$, such that the following diagram commutes:

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[dr, "\mathsf{id}_A"'] & B \arrow[d, "g"] \arrow[dr, "\mathsf{id}_B"] & \\
& A \arrow[r, "f"'] & B
\end{tikzcd}$$

Example. In Sets, an isomorphism is a bijection.

Here is a translation table, between the different languages of set theory, category theory and Haskell.

Set	Categories	Haskell
function	morphism	program
set	object	type
singleton set	terminal object	unit type
Cartesian product	(Cartesian) product	product type
element	morphism $1 \to X$	—
predicate	—	—
bijection	isomorphism	—

Theorem 2. Let $A, B, C \in |\mathcal{C}|$. A triple $(C, \mathsf{fst}\colon C \to A, \mathsf{snd}\colon C \to B)$, is a product of A and B if there is an operation

$$\frac{f\colon D \to A \qquad g\colon D \to B}{\langle f, g\rangle\colon D \to C}$$

such that

$$\mathsf{fst} \circ \langle f, g\rangle = f, \quad \mathsf{snd} \circ \langle f, g\rangle = g, \quad \langle \mathsf{fst}, \mathsf{snd}\rangle = \mathsf{id}, \quad \langle f, g\rangle \circ h = \langle f \circ h, g \circ h\rangle.$$

Proof. The proof consist of the soundness ($\Rightarrow$) and completeness ($\Leftarrow$) directions.

($\Rightarrow$) We need to show the claimed identities. The first two are obvious by definition. The other two are by diagram chasing:

$$\begin{tikzcd}
& A\times B \arrow[ld, bend right, "\mathsf{fst}"'] \arrow[d, dashed, "\langle\mathsf{fst},\mathsf{snd}\rangle"] \arrow[rd, bend left, "\mathsf{snd}"] & \\
A & A\times B \arrow[l, "\mathsf{fst}"] \arrow[r, "\mathsf{snd}"'] & B
\end{tikzcd}
\qquad
\begin{tikzcd}
& D \arrow[ldd, bend right, "f\circ h"'] \arrow[d, "h"] \arrow[rdd, bend left, "g\circ h"] & \\
& C \arrow[ld, bend right, "f"'] \arrow[d, "\langle f,g\rangle"] \arrow[rd, bend left, "g"] & \\
A & A\times B \arrow[l, "\mathsf{fst}"] \arrow[r, "\mathsf{snd}"'] & B
\end{tikzcd}$$

The first identity holds because in the left diagram replacing $\langle \mathsf{fst}, \mathsf{snd}\rangle$ with id would produce a diagram, which still commutes, but $\langle \mathsf{fst}, \mathsf{snd}\rangle$ is unique, hence $\langle \mathsf{fst}, \mathsf{snd}\rangle = \mathsf{id}$.

The second identity holds analogously because by the second diagram, $\langle f, g\rangle \circ h$ satisfies the characteristic property of $\langle f \circ h, g \circ h\rangle$.

($\Leftarrow$) Suppose, conversely, the identities hold and for some $h\colon D \to C$ the diagram:

$$\begin{tikzcd}
& D \arrow[ld, bend right, "f"'] \arrow[d, "h"] \arrow[rd, bend left, "g"] & \\
A & C \arrow[l, "\mathsf{fst}"] \arrow[r, "\mathsf{snd}"'] & B
\end{tikzcd}$$

commutes. Then

$$h = \mathsf{id} \circ h = \langle \mathsf{fst}, \mathsf{snd}\rangle \circ\ h = \langle \mathsf{fst} \circ h, \mathsf{snd} \circ h\rangle = \langle f, g\rangle. \qquad \square$$

Products are defined *not uniquely*, but only *uniquely up to (a unique) isomorphism.* Let e.g. $(A \times A, \mathsf{fst}, \mathsf{snd})$ be a product of A, A. Then $(A \times A, \mathsf{snd}, \mathsf{fst})$ is also a product of A, A:

$$\mathsf{swap}_A\colon A \times A \xrightarrow{\langle \mathsf{snd},\mathsf{fst}\rangle} A \times A \qquad\qquad
\begin{tikzcd}
A\times A \arrow[r, bend left, "\mathsf{swap}"] & A\times A \arrow[l, bend left, "\mathsf{swap}"]
\end{tikzcd}$$

The pair $(\mathsf{swap}_A, \mathsf{swap}_{A,A})$ is an isomorphism of $A \times A$ and $A \times A$:

$$\begin{aligned}
\mathsf{swap} \circ \mathsf{swap} &= \langle \mathsf{snd}, \mathsf{fst}\rangle \circ \langle \mathsf{snd}, \mathsf{fst}\rangle \\
&= \langle \mathsf{snd} \circ\langle \mathsf{snd}, \mathsf{fst}\rangle, \mathsf{fst} \circ\langle \mathsf{snd}, \mathsf{fst}\rangle\rangle \\
&= \langle \mathsf{fst}, \mathsf{snd}\rangle = \mathsf{id}.
\end{aligned}$$

By using specific names $\times$, fst, snd throughout we stick to *selected (binary) products.* In *Set*, standardly

$$A \times B = \{\langle x, y\rangle \mid x \in A, y \in B\}, \qquad \mathsf{fst}\langle x, y\rangle = x, \qquad \mathsf{snd}\langle x, y\rangle = y.$$

But we could just as well define

$$A \times B = \{\langle y, x\rangle \mid x \in A, y \in B\}, \qquad \mathsf{fst}\langle y, x\rangle = x, \qquad \mathsf{snd}\langle y, x\rangle = y.$$

Theorem 3. Products (if they exists) are unique up to isomorphism.

Proof. Let $(A \times B, \mathsf{fst}, \mathsf{snd})$ be a product of A, B and let $(A \Box B, \mathsf{fst}', \mathsf{snd}')$ be another product. Then the following diagram commutes:

$$\begin{tikzcd}
 & A \times B \arrow[ddl, bend right, "\mathsf{fst}"'] \arrow[ddr, bend left, "\mathsf{snd}"] \arrow[d, "\langle \mathsf{fst},\mathsf{snd}\rangle"'] & \\
 & A \Box B \arrow[dl, bend right, "\mathsf{fst}'"'] \arrow[dr, bend left, "\mathsf{snd}'"] \arrow[d, "\langle \mathsf{fst}',\mathsf{snd}'\rangle"] & \\
A & A \times B \arrow[l, "\mathsf{fst}"] \arrow[r, "\mathsf{snd}"'] & B
\end{tikzcd}$$

Hence, $\overbrace{\langle \mathsf{fst}, \mathsf{snd}\rangle}^{f} \circ \overbrace{\langle \mathsf{fst}' \mathsf{snd}'\rangle}^{g} = \mathsf{id}$ (because both morphisms satisfy the same characteristic property). Because of symmetry, also $g \circ f = \mathsf{id}$. Hence (f, g) is an isomorphism between $A \times B$ and $A \Box B$. □

From Haskell's perspective × is a *type constructor*, and since Haskell supports user defined type constructors, we can introduce arbitrary many isomorphic products, e.g.

```haskell
data Prod a b = Prod a b

proj1 :: Prod a b -> a
proj1 (Prod x _) = x

proj2 :: Prod a b -> b
proj2 (Prod _ y) = y

pair :: (a -> b) -> (a -> c) -> a -> Prod b c
pair f g x = Prod (f x) (g x)
```

Coproducts are *dual* to products, which is explicit in the following definition.

Definition (Coproducts)**.** An object $A + B$ together with morphisms $\mathsf{inl}\colon A \to A + B$ and $\mathsf{inr}\colon B \to A + B$ called left and right *injections* is a *coproduct* of A and B if for any $f\colon A \to C$ and any $g\colon B \to C$, there is a unique morphism $[f, g]\colon A + B \to C$, such that the following diagram commutes:

$$\begin{tikzcd}
 & C & \\
A \arrow[ur, bend left, "f"] \arrow[r, "\mathsf{inl}"] & A + B \arrow[u, "{[f,g]}"] & B \arrow[ul, bend right, "g"'] \arrow[l, "\mathsf{inr}"']
\end{tikzcd}$$

Intuitively, $[f, g]$ is defined by case distinction: if we are on the left of $A + B$ then we apply f; if we are on the right of $A + B$ then we apply g.

Example.

- In Sets, $A + B$ is the disjoint union $A \uplus B = \{(0, a) \mid a \in A\} \cup \{(1, b) \mid b \in B\}$.

- In Cpo, coproducts $A + B$ are inherited from Sets, and $x \sqsubseteq y$ for $x, y \in A + B$ iff both x and y are either in A or in B.

- In the category of relations, coproducts coincide with products and are thus called *biproducts*: $A + B$ is again the disjoint union of A and B, and $[r, s] \subseteq (A + B) \times C$ for $r \subseteq A \times C$, $s \subseteq B \times C$ is as follows: $(x, y) \in [r, s]$ iff $(x, y) \in r$ or $\langle x, y\rangle \in s$.

Dually to products we have a complete axiomatization for coproducts:

1. $[f, g] \circ \mathsf{inl} = f$;
2. $[f, g] \circ \mathsf{inr} = g$;
3. $[\mathsf{inl}, \mathsf{inr}] = \mathsf{id}$;
4. $h \circ [f, g] = [h \circ f, h \circ g]$.

Now, if we want to define a morphism $f \colon A + B \to C$, it suffices to define compositions $f \circ \mathsf{inl} \colon A \to C$ and $g \circ \mathsf{inr} \colon B \to C$, for f is uniquely determined by them: $f = f \circ [\mathsf{inl}, \mathsf{inr}] = [f \circ \mathsf{inl}, f \circ \mathsf{inr}]$. This justifies definitions "by case distinction": to define $f \colon A + B \to C$, we can write equations like $f(\mathsf{inl}\, a) = h(a)$ and $f(\mathsf{inr}\, b) = u(b)$, meaning $f \circ \mathsf{inl} = h$ and $g \circ \mathsf{inr} = u$, which, in turn, is the same as to define f.

Definition (Dual Category)**.** Given a category $\mathcal{C}$, the *dual category* $\mathcal{C}^{\mathsf{op}}$ is defined as follows:

- $|\mathcal{C}^{\mathsf{op}}| = |\mathcal{C}|$;
- $\mathcal{C}^{\mathsf{op}}(X, Y) = \mathcal{C}(Y, X)$.

Example. Let $\mathcal{C}$ be a poset category, i.e. $\mathcal{C}(X, Y) = \{\star\}$ iff $X \leqslant Y$. Then $\mathcal{C}^{\mathsf{op}}$ is the dually ordered poset: $\mathcal{C}^{\mathsf{op}}(X, Y) = \{\star\}$ iff $X \geqslant Y$.

For example, we now can formally state that products are dual to coproducts.

Proposition. For every $\mathcal{C}$, a binary product $\mathcal{C}^{\mathsf{op}}$ is a binary coproduct of $\mathcal{C}^{\mathsf{op}}$.

2.1.2 Functors and Monads

Definition (Functor)**.** A *(covariant) functor* between categories $\mathcal{C}$ and $\mathcal{D}$ is a correspondence sending any $A \in |\mathcal{C}|$ to $FA \in |\mathcal{D}|$ and any $f \in \mathcal{C}(A, B)$ to $Ff \in \mathcal{D}(FA, FB)$ in such a way that:

$$F(\mathsf{id}_A) = \mathsf{id}_{FA}, \qquad\qquad F(f \circ g) = (Ff) \circ (Fg).$$

Example (Forgetful Functor)**.** Forgetful functor is an informal concept: this is a functor that "forgets" some information about the category. One example is

$$\begin{aligned} &G \colon \mathrm{Cpo} \to \mathrm{Set} \\ &G(A, \sqsubseteq) = A \\ &G(f) = f \end{aligned}$$

G is a typical name for forgetful functors (to remember: for$\underline{\text{G}}$etful).

Example (Endofunctor)**.** An *endofunctor* is a functor from a category into itself. E.g.,

$$\begin{aligned}&F\colon \mathrm{Set} \to \mathrm{Set}\\&FX = X + E\\&(Ff)(\mathsf{inl}\,x) = \mathsf{inl}(fx)\\&(Ff)(\mathsf{inr}\,e) = \mathsf{inr}(e)\end{aligned}$$

Example (Finite Lists)**.** Another endofunctor over *Set*:

$$\begin{aligned}&F\colon \mathrm{Set} \to \mathrm{Set}\\&FX = [X] \quad \text{(finite lists over } X\text{)}\\&(Ff)[x_1, \dots, x_n] = [fx_1, \dots, fx_n]\end{aligned}$$

Definition (Contravariant Functor)**.** A functor $F\colon \mathcal{C}^{\mathsf{op}} \to \mathcal{D}$ is said to be a *contravariant functor* from $\mathcal{C}$ to $\mathcal{D}$.

Small categories themselves form a category with finite products: the final object is the category of one object and one arrow, and a product of categories $\mathcal{C}$ and $\mathcal{D}$ is the category $\mathcal{C}\times\mathcal{D}$ with

- $|\mathcal{C} \times \mathcal{D}| = |\mathcal{C}| \times |\mathcal{D}|$,
- $(\mathcal{C} \times \mathcal{D})((X, Y), (X', Y')) = \mathcal{C}(X, X') \times \mathcal{D}(Y, Y')$.

The category of all categories is not a category, more precisely, the locally small categories do not form a locally small category (but they form a category in a higher sense). Still, products of locally small categories make perfect sense regardless of this issue.

Definition (Bi-Functor)**.** A *bifunctor* is a functor $\mathcal{C} \times \mathcal{D} \to \mathcal{E}$ for which one also uses the notation $F(A, B)$ instead of $F(A \times B)$ and $F(f, g)$ instead of $F(f \times g)$.

Natural transforamtions ξ between bifunctors more explicitly satifying the following condition:

$$\begin{tikzcd}
F(A,B) \arrow[r, "\xi_{A,B}"] \arrow[d, "F(f,g)"'] & G(A,B) \arrow[d, "G(f,g)"] \\
F(A' \times B') \arrow[r, "\xi_{A',B'}"] & G(A' \times B')
\end{tikzcd}$$

for any $f\colon A \to A'$ and $g\colon B \to B'$.

Example (Product Functor)**.** Let $\mathcal{C}$ have binary products. Then $F\colon \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ sending A, B to $A \times B$ is a bi-functor with $F(f, g) = f \times g$.

Example (Hom-Functor)**.** The *hom-functor* is the bi-functor $\mathsf{Hom}(\text{–}, \text{–})\colon \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathbf{Set}$.

Now, instead of saying that $\alpha\colon F \to G$ is a natural transformation, one often says that a family $\alpha_A\colon FA \to GA$ is *natural in* A, e.g. for bi-functors, $F\colon \mathcal{C} \times \mathcal{D} \to \mathcal{E}$, naturality of $\alpha_{A,B}\colon F(A \times B) \to G(A \times B)$ in A and B. Another example: associativity $\alpha_{A,B,C}\colon A \times (B \times C) \to (A \times B) \times C$ is natural in A, B, C.

Definition (Monad/Kleisli Triple)**.** A *Monad* in a category $\mathcal{C}$ is given by a triple $(T, \eta, \_^\star)$ (*Kleisli triple*) where

- $T\colon |\mathcal{C}| \to |\mathcal{C}|$,
- η is a family $(\eta_X\colon X \to TX)_{X\in|\mathcal{C}|}$ (*unit*),
- for any $f\colon A \to TB$, $f^\star\colon TA \to TB$ (*(Kleisli) lifting*)

and the following laws are satisfied:

$$\eta^\star = \mathsf{id}, \qquad f^\star\eta = f, \qquad (f^\star g)^\star = f^\star g^\star.$$

Example (Exception monad)**.** $TX = X + E$ is a monad with:

$$\eta_X(a) = \mathsf{inl}\, a \qquad f^\star(\mathsf{inl}\, a) = fa \qquad f^\star(\mathsf{inr}\, e) = \mathsf{inr}\, e$$

This works in any category $\mathcal{C}$ with coproducts, $TX = X + E$ extends to a monad under the following definitions:

$$\begin{aligned} \eta_X &= \mathsf{inl}\colon X \to X + E \\ f^\star &= [f, \mathsf{inr}]\colon X + E \to Y + E \text{ where } f\colon X \to Y + E \end{aligned}$$

Intuitively, f is a function, which may *raise* an exception, and $f^\star$ completes the definition of f by the clause: “if an exception has already been raised before, pass it as the result”.

It is easy to check that T from a monad $(T, \eta, -^\star)$ is a functor. We call it the *functorial part* of the monad.

Definition (Kleisli Category)**.** Given a monad T over a category $\mathcal{C}$, the *Kleisli category* $\mathcal{C}_T$ of T is defined as follows:

- $|\mathcal{C}_T| = |\mathcal{C}|$;
- $\mathcal{C}_T(A, B) = \mathcal{C}(A, TB)$;
- identity morphisms in $\mathcal{C}_T$ are $\eta_X \in \mathcal{C}_T(X, X) = \mathcal{C}(X, TX)$;
- composition of $f\colon A \to TB$ and $g\colon B \to TC$ is *Kleisli composition*: $g^\star f\colon A \to TC$.

Theorem 4. $\mathcal{C}_T$ is a category:

1. $\eta^\star f = \mathsf{id} \circ f = f$
2. $f^\star\eta = f$
3. $f^\star(g^\star h) = (f^\star g^\star)h = (f^\star g)^\star h$

Let $f \times g$ denote $\langle f \circ \mathsf{fst}, g \circ \mathsf{snd}\rangle\colon A \times B \to A' \times B'$ where $f\colon A \to A'$ and $g\colon B \to B'$. It is easy to check some obvious properties of this notation like $(f \times g) \circ (f' \times g') = (f \circ f') \times (g \circ g')$ and $(f \times g) \circ \langle f', g'\rangle = \langle f \circ f', g \circ g'\rangle$.

Let

$$\begin{aligned} \alpha_{A,B,C} &= \langle \mathsf{id} \times \mathsf{fst}, \mathsf{snd} \circ \mathsf{snd}\rangle\colon\ A \times (B \times C) \to (A \times B) \times C; \\ \alpha^{-1}_{A,B,C} &= \langle \mathsf{fst} \circ \mathsf{fst}, \mathsf{snd} \times \mathsf{id}\rangle\colon\ (A \times B) \times C \to A \times (B \times C). \end{aligned}$$

Obviously, α and α^{-1} are mutualy inverse. Analogously, we define *unitors*:

$$\lambda_A = \big(A \times 1 \xrightarrow{\mathsf{fst}} A\big), \qquad \rho_A = \big(1 \times A \xrightarrow{\mathsf{snd}} A\big)$$

for which $\lambda^{-1}_A = \langle \mathsf{id}_A, !\rangle$, $\rho^{-1}_A = \langle !, \mathsf{id}_A\rangle$.

Theorem 5 (Mac Lane's Coherence Theorem[1])**.** Any diagram with labels composed from $\mathsf{id}, \times, \alpha, \alpha^{-1}, \lambda, \lambda^{-1}, \rho, \rho^{-1}$ commutes.

2.1.3 Natural Transformations: Relating Functors

Associativity morphisms $\alpha_{A,B,C}$ are examples of natural transformations, which are a categorical formalization of parametric dependency.

Definition (Natural Transformation)**.** Let $\mathcal{C}, \mathcal{D}$ be categories and $F, G \colon \mathcal{C} \to \mathcal{D}$ be functors. A *natural transformation* $\vartheta \colon F \to G$ is a *family* of morphisms in $\mathcal{D}$:

$$(\vartheta_C \colon FC \to GC)_{C \in |\mathcal{C}|},$$

such that, for any $f \colon C \to C'$ in $\mathcal{C}$, the following (naturality) diagram commutes:

$$\begin{tikzcd}
FC \arrow[r, "\vartheta_C"] \arrow[d, "Ff"'] & GC \arrow[d, "Gf"] \\
FC' \arrow[r, "\vartheta_{C'}"] & GC'
\end{tikzcd}$$

The morphisms $\vartheta_C : FC \to GC$ are called *components* of $\vartheta \colon F \to G$.

Intuitively, natural transformations are such morphisms $\vartheta_C : FC \to GC$ that do not use any information about C. Instead of saying "$\vartheta \colon F \to G$ is a natural transformation" one often uses equivalent formulation "$\vartheta_C \colon FC \to GC$ is a morphism natural in C".

Semantically, naturality corresponds to a specific form of *parametric polymorphism*. Haskell functions are automatically polymorphic in the corresponding *type variables*, but not necessarily natural. E.g. Haskell's function

```
reverse :: [a] -> [a]
```

for list reversal is polymorphic in `a` as well as natural it in the categorical sense, but

```
sort :: Ord a => [a] -> [a]
```

for sorting lists is not natural, which is indicated by the *type constraint* ”`Ord a =>`” telling that sorting is not independent of the type `a` – the result depends on the fact that `a` is an ordered type and on that how it is ordered.

Another example of a natural transformation:

```
maybeToList :: Maybe a -> [a]
maybeToList (Just a) = [a]
maybeToList Nothing  = []
```

[1] simplified version

Definition. For any functor F and natural transformation $\vartheta\colon G \to H$ we define natural transformations $\vartheta_F : GF \to HF$ and $F\vartheta\colon FG \to FH$ as follows:

$$(\vartheta_F)_X = \vartheta_{FX}$$
$$(F\vartheta)_X = F(\vartheta_X).$$

(Easy) exercise: show that ϑ_F and $F\vartheta$ are indeed natural transformations.

Remark A natural transformation $F \xrightarrow{\xi} G$ is often drawn as $$\begin{tikzcd} \mathcal{C} \arrow[r, bend left, "F"] \arrow[r, bend right, "G"'] \arrow[r, phantom, "\Downarrow \xi"] & \mathcal{D} \end{tikzcd}$$. This would be consistent with the notation $\xi\colon F \Rightarrow G$, which is often used for natural transformations. We simply write $\xi\colon F \to G$ instead, for, after all, natural transformations are just morphisms in the functor category $[F, G]$.

Theorem 6. Cat is defined as follows:

- $|\mathsf{Cat}|$ are small Categories $\mathcal{C}$ (that is, those for which $|\mathcal{C}|$ is a set).
- $\mathsf{Cat}(\mathcal{C}, \mathcal{D})$ is the class of all functors from $\mathcal{C}$ to $\mathcal{D}$.

Cat is itself a category with $\mathsf{id}\colon \mathcal{C} \to \mathcal{C}$ being the identity functor and $F \circ G$ being functor composition $\mathcal{C} \xrightarrow{G} \mathcal{D} \xrightarrow{F} \mathcal{E}$.

Proof. trivial. □

Theorem 7. Given two categories $\mathcal{C}$ and $\mathcal{D}$, $[\mathcal{C}, \mathcal{D}]$, defined as follows:

- $|[\mathcal{C}, \mathcal{D}]|$ are functors from $\mathcal{C}$ to $\mathcal{D}$;
- $[\mathcal{C}, \mathcal{D}](F, G)$ are natural transformations $\xi\colon F \to G$.

is again a category.

Proof.

1. $\mathsf{id} \circ \xi = \xi$: For any $f\colon A \to B$

$$\begin{tikzcd}
FA \arrow[r, "\xi_A"] \arrow[rr, bend left, "\xi_A"] \arrow[d, "Ff"] & GA \arrow[r, "\mathsf{id}_A"] \arrow[d, "Gf"] & GA \arrow[d, "Gf"] \\
FB \arrow[r, "\xi_B"] \arrow[rr, bend right, "\xi_B"'] & GB \arrow[r, "\mathsf{id}_B"] & GB
\end{tikzcd}$$

2. $\xi \circ \mathsf{id} = \xi$
3. $\xi \circ (\theta \circ \sigma) = (\xi \circ \theta) \circ \sigma$

Properties 2 and 3 are analogous to proof. □

Definition (Vertical composition)**.** Pointwise composition of natural transformations ($(\xi \circ \theta)_A = \xi_A \circ \theta_A$) is called *vertical composition*:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left=60, "F"{name=F}] \arrow[r, "G"{name=G}] \arrow[r, bend right=60, "H"'{name=H}] & \mathcal{D}
\arrow[Rightarrow, from=F, to=G, "\theta"] \arrow[Rightarrow, from=G, to=H, "\xi"]
\end{tikzcd}$$

Definition (Horizontal composition)**.** Given $\xi\colon F \to F'$ and $\theta\colon G \to G'$,

$$\xi \circ \theta\colon GF \to G'F'$$

is defined by the diagram:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left=40, "F"{name=F}] \arrow[r, bend right=40, "F'"'{name=F2}] & \mathcal{D} \arrow[r, bend left=40, "G"{name=G}] \arrow[r, bend right=40, "G'"'{name=G2}] & \mathcal{E}
\arrow[Rightarrow, from=F, to=F2, "\xi"] \arrow[Rightarrow, from=G, to=G2, "\theta"]
\end{tikzcd}$$

Notation. Given $\xi\colon F \to G$, we can form $H\xi\colon HF \to HG$ and $\xi_U\colon FU \to GU$, also written $\xi U\colon FU \to GU$, by putting

$$(H\xi)_A = H(\xi_A) \qquad (\xi_U)_A = \xi_{UA}$$

Proposition. Given $\xi\colon F \to F'$ and $\theta\colon G \to G'$ then $\xi \circ \theta = (\theta_{F'}) \circ (G\xi)$

Example. $\mathsf{elems}_A : [A] \to \mathcal{P}(A)$ defined as follows:

$$\mathsf{elems}_A([l_1, \ldots, l_2]) = \{l_1, \ldots, l_n\}$$

yields a natural transformation $\mathsf{elems}\colon [\ ] \to \mathcal{P}$ of endofunctors over Sets. Naturality: Let $f\colon A \to B$. Then

$$(\mathcal{P}f) \circ \mathsf{elems} \circ ([l_1, \ldots, l_n]) = (\mathcal{P}f) \circ \{l_1, \ldots, l_n\} = \{f(l_1), \ldots, f(l_n)\}.$$

On the other hand:

$$(\mathsf{elems}_B \circ [f])[l_1, \ldots, l_n] = \mathsf{elems}_B([f(l_1), \ldots, f(l_n)]) = \{f(l_1), \ldots, f(l_n)\}.$$

We now can give a new (equivalent) definition of a monad.

Definition (Monad)**.** A monad on a category $\mathcal{C}$ consists of an *endofunctor* $T : \mathcal{C} \to \mathcal{C}$, and natural transformations

$$\underbrace{\eta\colon \mathrm{Id} \to T}_{unit}, \qquad \underbrace{\mu\colon TT \to T}_{multiplication}$$

making the diagrams

$$\begin{tikzcd}
TTTX \arrow[r, "\mu_{TX}"] \arrow[d, "T\mu_X"'] & TTX \arrow[d, "\mu_X"] \\
TTX \arrow[r, "\mu_X"] & TX
\end{tikzcd}
\qquad
\begin{tikzcd}
TX \arrow[r, "\eta_{TX}"] \arrow[dr, "\mathsf{id}_{TX}"'] & TTX \arrow[d, "\mu_X"] & TX \arrow[l, "T\eta_X"'] \arrow[dl, "\mathsf{id}_{TX}"] \\
& TX &
\end{tikzcd}$$

commute, i.e. the equations

$$\mu \circ \mu_T = \mu \circ T\mu \qquad\qquad \mu \circ \eta_T = \mathsf{id} = \mu \circ T\eta.$$

Proposition. Given a Kleisli-Triple $(T', \eta', \_^\star)$ satisfying the monad laws, one obtains a monad in the sense defined above in the following way:

$$\begin{aligned} Tf &= (\eta \circ f)^\star \quad \text{for } f\colon X \to Y \\ TX &= T'X \\ \eta_X &= \eta'_X \\ \mu_X &= (\mathsf{id}_{TX})^\star \end{aligned}$$

2.1.4 Examples of Monads

IO Monad

```
instance Monad IO
getLine   :: IO String
putStrLn  :: String -> IO ()

do x <- getLine; putStrLn $ "yes, exactly, " ++ x ++ "!"
```

Rough intuition: `IO` $A = \mathit{World} \to A \times \mathit{World}$:

$$\begin{aligned} \texttt{getLine}\colon 1 &\to (\mathit{World} \to (\mathit{String} \times \mathit{World})) \\ \texttt{getLine}(x)(w) &= \langle \mathsf{receiveLineFromWorld}(w), w \rangle \\ \texttt{putString}(s)(w) &= \langle 1, \mathsf{sendLineToWorld}(s, w) \rangle \end{aligned}$$

State Monad

$$TX = (X \times S)^S$$

This works in Sets, Cpos and more generally in Cartesian closed categories.

$$\begin{aligned} \eta_X &\colon X \to (X \times S)^S & (f\colon X \to (Y \times S)^S)^\star &\colon (X \times S)^S \to (Y \times S)^S \\ \eta_X(x)(s) &= \langle x, s \rangle & f^\star(p)(s) &= \mathsf{let}\langle x, s' \rangle = p(s) \,\mathsf{in}\, f(x)(s') \end{aligned}$$

where

$$\mathsf{let}\langle x, y \rangle = p \,\mathsf{in}\, q = \mathsf{let}\, z = p \,\mathsf{in}\, q[\mathsf{fst}\, z/x, \mathsf{snd}\, z/y]$$

The state monad supports the following operations:

$$\begin{aligned} \mathrm{put}\colon S &\to T1 & \mathrm{put}(s)(s') &= (*, s) \\ \mathrm{get}\colon 1 &\to TS & \mathrm{get}(*)(s) &= (s, s) \end{aligned}$$

Example (Writer Monand)**.**

$$TX = M \times X \quad \text{(where } M \text{ is a Monoid)}$$

Example (Reader Monad)**.**

$$TX = X^S$$

The Reader Monad is a submonad of the State monad:

$$\alpha_X \colon X^S \to (X \times S)^S$$
$$\alpha_X(p)(s) = (p(s), s)$$

Theorem 8. $TX = X^S$ is a monad.

Continuation Monad In Sets: $TX = \underbrace{(X \to R)}_{\text{Continuation}} \to \underbrace{R}_{\text{Result}}$

$$\eta_X(x) = \lambda k.\ k(x)$$
$$(f \colon X \to (R^Y \to R))^\star(p \colon R^X \to R) = \lambda k : Y \to R.\ p(\underbrace{\lambda x.f(x)(k)}_{X \to R})$$

The following lemma helps to prove that the continuation monad is indeed a monad in an abstract way.

Lemma. Let $F \colon \mathcal{C} \to \mathcal{D}$ be a functor and let T be a map $|\mathcal{C}| \to |\mathcal{C}|$. Suppose that for any $X, Y \in |\mathcal{C}|$, the hom-sets $\mathsf{Hom}(X, TY)$ and $\mathsf{Hom}(FX, FY)$ are isomorphic naturally in X. Then T is a monad with the following induced structure

$$\eta = \check{\mathsf{id}} \qquad\qquad f^\star = \widecheck{\hat{f}\,\hat{\mathsf{id}}}$$

where $\hat{f} \colon FX \to FY$ and $\check{g} \colon X \to TY$ are the obvious isomorphic images of $f \colon X \to TY$ and $g \colon FX \to FY$ correspondingly.

Moreover, the Kleisli category of T is isomorphic to the full subcategory of $\mathcal{D}$ over the objects of the form FX.

Proof. The naturality condition means precisely that $\hat{f}(Fh) = \widehat{fh}$ for any $h \colon X \to Y$ and $f \colon Y \to TY$. This entails that $\widecheck{g(Fh)} = \check{g}h$ for $g = \check{f}$ and moreover,

$$f^\star g = \widecheck{\hat{f}\,\hat{\mathsf{id}}}g = \widecheck{\hat{f}\,\hat{\mathsf{id}}\,Fg} = \widecheck{\hat{f}\,\widehat{\mathsf{id}\,g}} = \widecheck{\hat{f}\,\hat{g}}\ .$$

Therefore,

$$\eta^\star = \widecheck{\hat{\check{\mathsf{id}}}\,\hat{\mathsf{id}}} = \check{\hat{\mathsf{id}}} = \mathsf{id}$$

$$f^\star\eta = \widecheck{\hat{f}\,\hat{\eta}} = \check{\hat{f}} = f$$

$$(f^\star g)^\star = \widecheck{(\hat{f}\,\hat{g})\,\hat{\mathsf{id}}} = \widecheck{\hat{f}\,(\hat{g}\,\hat{\mathsf{id}})} = f^\star\,\widecheck{\hat{g}\,\hat{\mathsf{id}}} = f^\star\,g^\star,$$

and we are done. □

This can be instantiated as follows.

Example. For the state monad $TX = (X \times S)^S$, $\mathsf{Hom}_{\mathcal{C}}(X, TY) \cong \mathcal{C}(X \times S, Y \times S)$.

For the continuation monad $TX = (X \to R) \to R$, $\mathsf{Hom}_{\mathcal{C}}(X, TY) \cong \mathsf{Hom}_{\mathcal{C}^{\mathsf{op}}}(R^X, R^Y) = \mathsf{Hom}_{\mathcal{C}}(R^Y, R^X)$.

2.1.5 Cartesian Closure

Definition (Cartesian Closure)**.** Let $\mathcal{C}$ be a Cartesian category. Given $X, Y \in |\mathcal{C}|$, $X^Y \in |\mathcal{C}|$, together with a morphism $\mathsf{ev}\colon X^Y \times Y \to X$ is called an *exponential object*, if for every $f\colon Z \times Y \to X$ there is unique $g\colon Z \to X^Y$, such that the following diagram commutes:

$$\begin{tikzcd}
Z \times Y \arrow[r, "g \times \mathsf{id}"] \arrow[dr, "f"'] & X^Y \times Y \arrow[d, "\mathsf{ev}"] \\
& X
\end{tikzcd}$$

The forall-exists clause can be interpreted as existence of an operation $\mathcal{C}(Z \times Y, X) \to \mathcal{C}(Z, X^Y)$, which is standardly called curry. The category $\mathcal{C}$ is called *Cartesian closed* if all exponentials X^Y exist.

Proposition. In any CCC $\mathcal{C}$, A^B extends to a bi-functor $(\text{–})^{(\text{–})}\colon \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathcal{C}$ sending $f\colon A' \to A$ and $g\colon B \to B'$ to

$$\mathsf{curry}(B^A \times A' \xrightarrow{\mathsf{id} \times f} B^A \times A \xrightarrow{\mathsf{ev}} B \xrightarrow{g} B')\colon B^A \to B'^{A'}.$$

Proposition. In any CCC curry and $\mathsf{uncurry}$ are natural in all parameters.

2.2 Tensorial Strength

Suppose that we want to add a new type former to the type grammar of PCF

$$A, B, C, \ldots ::= 1 \mid A \times B \mid A \to B \mid TA$$

We can interpret such a language in any CCC with a monad T on it, with suitable carriers $[\![Bool]\!]$, $[\![Nat]\!]$, and a fixpoint operator $\mathit{fix}\colon (A \to A) \to A$. Recall the semantics of types:

- $[\![1]\!] = 1$;
- $[\![A \times B]\!] = [\![A]\!] \times [\![B]\!]$;
- $[\![A \to B]\!] = [\![B]\!]^{[\![A]\!]}$.

We add $[\![TA]\!] = T[\![A]\!]$ to that. Now, the semantics of a term in context $\Gamma \vdash t\colon A$ with $\Gamma = (x_1\colon A_1, \ldots, x_n\colon A_n)$ must be a morphism $[\![A_1]\!] \times \ldots \times [\![A_n]\!] \to [\![A]\!]$. This works alright, and we could also incorporate the do-notation in the language:

$$\frac{\Gamma \vdash p\colon TA \qquad \Gamma, x\colon A \vdash q\colon TB}{\Gamma \vdash \mathsf{do}\, x \leftarrow p; q\colon TB}$$

Here we have:

$$\begin{aligned} f &= [\![\Gamma \vdash p\colon TA]\!]\colon [\![\Gamma]\!] \to T[\![A]\!] \\ g &= [\![\Gamma, x\colon A \vdash q\colon TB]\!]\colon [\![\Gamma]\!] \times [\![A]\!] \to T[\![B]\!] \end{aligned}$$

from which we expect to obtain:

$$[\![\Gamma \vdash \mathsf{do}\, x \leftarrow p; q\colon TB]\!]\colon [\![\Gamma]\!] \to T[\![B]\!]$$

We would expect to have

$$[\![\Gamma]\!] \xrightarrow{\langle \mathsf{id}, f\rangle} [\![\Gamma]\!] \times T[\![A]\!] \xrightarrow{?} T([\![\Gamma]\!] \times [\![A]\!]) \xrightarrow{g^\star} T[\![B]\!]$$

That is, we need means to incorporate the context Γ into a computation of type A.

2.2.1 Strong Monads

We arrive at the following notion.

Definition (Tensorial Strength)**.** A strong functor is a functor $F\colon \mathcal{C} \to \mathcal{D}$ between Cartesian categories $\mathcal{C}$ and $\mathcal{D}$, plus *strength*, which is a natural transformation $\tau_{A,B}\colon A\times FB \to F(A\times B)$, such that

$$\begin{tikzcd} 1 \times FX \arrow[r, "\mathsf{snd}"] \arrow[d, "\tau"'] & FX \\ F(1 \times X) \arrow[ur, "F\,\mathsf{snd}"'] & \end{tikzcd} \qquad \begin{tikzcd} (X \times Y) \times FZ \arrow[rr, "\tau"] \arrow[d, "assoc"'] & & F((X \times Y) \times Z) \arrow[d, "F assoc"] \\ X \times (Y \times FY) \arrow[r, "X\times\tau"] & X \times F(Y \times Z) \arrow[r, "\tau"] & F(X \times (Y \times Z)) \end{tikzcd}$$

Strong natural transformations are those that preserve strength in the obvious sense. Given a strong functor (F, τ), note that $(\mathrm{Id}, \mathsf{id}\colon X \times Y \to X \times Y)$ and $(FF, (F\tau)\tau\colon X \times FFY \to FF(X \times Y))$ are again strong functors.

Now, a monad is *strong* if it is strong as a functor and η, μ are strong natural transformations, concretely,

$$\begin{tikzcd} X \times Y \arrow[r, "\eta"] \arrow[d, "\mathsf{id}\times\eta"'] & T(X \times Y) \\ X \times TY \arrow[ur, "\tau"'] & \end{tikzcd} \qquad \begin{tikzcd} X \times TTY \arrow[rr, "\mathsf{id}\times\mu"] \arrow[d, "\tau"'] & & X \times TY \arrow[d, "\tau"] \\ T(X \times TY) \arrow[r, "T\tau"] & TT(X \times Y) \arrow[r, "\mu"] & T(X \times Y) \end{tikzcd}$$

The reason why we do not see strength when programming in Haskell is because Haskell functors $F\colon \mathcal{C} \to \mathcal{C}$ are indeed natural transformations $A^B \to FA^{FB}$ (as opposed to categorical functors $\mathsf{Hom}(A, B) \to \mathsf{Hom}(FA, FB)$). Categorically, this is in fact, a quite specific condition.

Definition (Functorial Strength)**.** An endofunctor $F\colon \mathcal{C} \to \mathcal{C}$ on a CCC $\mathcal{C}$ is functorially strong, if it comes with a *functorial strength*, i.e. a family of morphisms

$$\rho_{A,B}\colon B^A \to FB^{FA},$$

such that

$$\begin{tikzcd}
\mathsf{Hom}(1 \times A, B) \arrow[r, "\cong"] \arrow[d, "\mathsf{curry}"'] & \mathsf{Hom}(A, B) \arrow[r, "F"] & \mathsf{Hom}(FA, FB) \arrow[r, "\cong"] & \mathsf{Hom}(1 \times FA, FB) \arrow[d, "\mathsf{curry}"] \\
\mathsf{Hom}(1, B^A) \arrow[rrr, "\mathsf{Hom}(1, \rho_{A,B})"'] & & & \mathsf{Hom}(1, FB^{FA})
\end{tikzcd}$$

Moreover, ρ must respect internal units ($\mathsf{curry}(\mathsf{snd})\colon 1 \to A^A$) and composition ($B^A \times C^B \to C^A$) in an obvious sense.

Analogously, we can internalise natural transformations and define "functorialy strong monad" as those functorially strong functors, for which there are internalized version of η and μ.

It turns out however that tensorial strength and functorial strength are equivalent:

$$\begin{aligned}
\tau_{A,B} &= \mathsf{uncurry}\big(A \xrightarrow{\mathsf{curry\,id}} (A \times B)^B \xrightarrow{\rho} T(A \times B)^{TB}\big),\\
\rho_{A,B} &= \mathsf{curry}\big(B^A \times TA \xrightarrow{\tau} T(B^A \times A) \xrightarrow{T\,\mathsf{ev}} TB\big).
\end{aligned}$$

Example. Every endofunctor and every monad on *Set* are strong with the functorial strength being just the functorial action, because there is no distinction between hom-sets $\mathsf{Hom}(A, B)$ and exponentials B^A. Hence $\tau_{A,B}(x \in A, p \in TB) = (T\lambda y.\, \langle x, y\rangle)(p)$ (now we see, what this expression actually means!)

Every monad on predomains is thus also strong – this amounts to verifying that the above τ is continuous.

Categorically, the right setup for these considerations is *enriched categories*. These generalize standard categories by replacing *hom-sets* with *hom-objects* of a yet another category $\mathcal{V}$, in which the original category is said to be *enriched*. This produces the whole spectrum of derived notions: $\mathcal{V}$-functors, $\mathcal{V}$-natural transormations, $\mathcal{V}$-monads, etc. From this perspective our categories are *Set*-categories, i.e. categories enriched in *Set*. Every Cartesian closed category can be regarded as enriched over itself, because we can use exponentials A^B instead of hom-sets $\mathsf{Hom}(B, A)$. In that sense strong functors turn out to be precisely the enriched functors and strong monads turn out to be precisely the enriched monads. As a slogan: *in CCC strength is equivalent to enrichment*[2].

Is there non-strong monads? They are not easy to meet in the wild.

Example (Non-Strong Monad)**.** In the category of *two-sorted sets* $Set^2 = Set \times Set$ the monad $(X, Y) \mapsto (X, Y + X)$ is not strong.

2.2.2 Commutative Monads

We can classify computational effects according the equations they satisfy. Recall that the lifting monad satisfies the commutativity property:

$$\mathsf{let}\, x = p \,\mathsf{in}\, \mathsf{let}\, y = q \,\mathsf{in}\, \lfloor \langle x, y\rangle \rfloor = \mathsf{let}\, y = q \,\mathsf{in}\, \mathsf{let}\, x = p \,\mathsf{in}\, \lfloor \langle x, y\rangle \rfloor,$$

[2] Anders Kock. "Strong Functors and Monoidal Monads". In: *Archiv der Mathematik* 23.1 (1972), pp. 113–120.

where $\hat{\tau}_{A,B}\colon TA \times B \to T(A \times B)$ is the following dual of $\tau_{A,B}$:

$$TA \times B \xrightarrow{\langle \mathsf{snd}, \mathsf{fst}\rangle} B \times TA \xrightarrow{\tau_{B,TA}} T(B \times A) \xrightarrow{T\langle \mathsf{snd}, \mathsf{fst}\rangle} T(A \times B).$$

Definition (Commutative Monad)**.** A strong monad T is commutative if

$$\begin{tikzcd}
TA \times TB \arrow[r, "\tau"] \arrow[d, "\hat{\tau}"'] & T(TA \times B) \arrow[r, "T\hat{\tau}"] & TT(A \times B) \arrow[dd, "\mu"] \\
T(A \times TB) \arrow[d, "T\tau"'] & & \\
TT(A \times B) \arrow[rr, "\mu"] & & T(A \times B)
\end{tikzcd}$$

This is the same as claiming

$$\mathsf{do}\, x \leftarrow p;\ \mathsf{do}\, y \leftarrow q;\ \mathsf{return}\langle x, y\rangle = \mathsf{do}\, y \leftarrow q;\ \mathsf{do}\, x \leftarrow p;\ \mathsf{return}\langle x, y\rangle$$

in the do-notation style, with x and y not occurring in p and q. Let us check it. Let

$$f = [\![\Gamma \vdash p\colon TA]\!], \qquad g = [\![\Gamma \vdash q\colon TB]\!],$$

and show that

$$\begin{aligned}[][\![\Gamma \vdash \mathsf{do}\, x \leftarrow p;\ \mathsf{do}\, y \leftarrow q;\ \mathsf{return}\langle x, y\rangle\colon T(A \times B)]\!]&\\ = [\![\Gamma \vdash \mathsf{do}\, y \leftarrow q;\ \mathsf{do}\, x \leftarrow p;\ \mathsf{return}\langle x, y\rangle\colon T(A \times B)]\!]&\end{aligned}$$

assuming that the underlying monad T is commutative. The left-hand side evaluates to

$$w = \big([\![\Gamma]\!] \xrightarrow{\langle \mathsf{id}, f\rangle} [\![\Gamma]\!] \times T[\![A]\!] \xrightarrow{\tau} T([\![\Gamma]\!] \times [\![A]\!]) \xrightarrow{h^\star} T([\![A]\!] \times [\![B]\!])\big)$$

where

$$h = \big([\![\Gamma]\!] \times [\![A]\!] \xrightarrow{g \times \mathsf{id}} T[\![B]\!] \times [\![A]\!] \xrightarrow{\hat{\tau}} T([\![B]\!] \times [\![A]\!]) \xrightarrow{T\langle \mathsf{snd}, \mathsf{fst}\rangle} T([\![A]\!] \times [\![B]\!])\big)$$

Analogously, the right-hand side evaluates to

$$w' = \big([\![\Gamma]\!] \xrightarrow{\langle \mathsf{id}, g\rangle} [\![\Gamma]\!] \times T[\![B]\!] \xrightarrow{\tau} T([\![\Gamma]\!] \times [\![B]\!]) \xrightarrow{(h')^\star} T([\![A]\!] \times [\![B]\!])\big)$$

where

$$h' = \big([\![\Gamma]\!] \times [\![B]\!] \xrightarrow{f \times \mathsf{id}} T[\![A]\!] \times [\![B]\!] \xrightarrow{\hat{\tau}} T([\![A]\!] \times [\![B]\!])\big)$$

So, we need to show that $w = w'$, i.e. that

$$(T\langle \mathsf{snd}, \mathsf{fst}\rangle \circ \hat{\tau} \circ (g \times \mathsf{id}))^\star \circ \tau \circ \langle \mathsf{id}, f\rangle = (\hat{\tau} \circ (f \times \mathsf{id}))^\star \circ \tau \circ \langle \mathsf{id}, g\rangle.$$

Indeed,

$$(T\langle \mathsf{snd}, \mathsf{fst}\rangle \circ \hat{\tau} \circ (g \times \mathsf{id}))^\star \circ \tau \circ \langle \mathsf{id}, f\rangle$$

$$
\begin{aligned}
&= \mu \circ T(T\langle \mathsf{snd}, \mathsf{fst}\rangle \circ \hat{\tau} \circ (g \times \mathsf{id})) \circ \tau \circ \langle \mathsf{id}, f\rangle && \text{// since } (-)^\star = \mu \circ T \\
&= \mu \circ TT\langle \mathsf{snd}, \mathsf{fst}\rangle \circ T\hat{\tau} \circ T(g \times \mathsf{id}) \circ \tau \circ \langle \mathsf{id}, f\rangle && \text{// functoriality of } T \\
&= T\langle \mathsf{snd}, \mathsf{fst}\rangle \circ \mu \circ T\hat{\tau} \circ T(g \times \mathsf{id}) \circ \tau \circ \langle \mathsf{id}, f\rangle && \text{// naturality of } \mu \\
&= T\langle \mathsf{snd}, \mathsf{fst}\rangle \circ \mu \circ T\hat{\tau} \circ \tau \circ (g \times \mathsf{id}) \circ \langle \mathsf{id}, f\rangle && \text{// naturality of } \tau \\
&= T\langle \mathsf{snd}, \mathsf{fst}\rangle \circ \mu \circ T\tau \circ \hat{\tau} \circ (g \times \mathsf{id}) \circ \langle \mathsf{id}, f\rangle && \text{// commutativity of } T \\
&= T\langle \mathsf{snd}, \mathsf{fst}\rangle \circ \mu \circ T\tau \circ \hat{\tau} \circ (g \times \mathsf{id}) \circ \langle g, f\rangle && \\
&= T\langle \mathsf{snd}, \mathsf{fst}\rangle \circ \mu \circ T\tau \circ \hat{\tau} \circ (\mathsf{id} \times f) \circ \langle g, \mathsf{id}\rangle && \\
&= \mu \circ TT\langle \mathsf{snd}, \mathsf{fst}\rangle \circ T\tau \circ \hat{\tau} \circ (\mathsf{id} \times f) \circ \langle g, \mathsf{id}\rangle && \text{// naturality of } \mu \\
&= \mu \circ T(T\langle \mathsf{snd}, \mathsf{fst}\rangle \circ \tau) \circ \hat{\tau} \circ (\mathsf{id} \times f) \circ \langle g, \mathsf{id}\rangle && \text{// functoriality of } T \\
&= \mu \circ T(\hat{\tau} \circ \langle \mathsf{snd}, \mathsf{fst}\rangle) \circ \hat{\tau} \circ (\mathsf{id} \times f) \circ \langle g, \mathsf{id}\rangle && \text{// defn. of } \hat{\tau} \\
&= \mu \circ T\hat{\tau} \circ T\langle \mathsf{snd}, \mathsf{fst}\rangle \circ \hat{\tau} \circ (\mathsf{id} \times f) \circ \langle g, \mathsf{id}\rangle && \text{// functoriality of } T \\
&= \mu \circ T\hat{\tau} \circ \tau \circ \langle \mathsf{snd}, \mathsf{fst}\rangle \circ (\mathsf{id} \times f) \circ \langle g, \mathsf{id}\rangle && \text{// defn. of } \hat{\tau} \\
&= \mu \circ T\hat{\tau} \circ \tau \circ (f \times \mathsf{id}) \circ \langle \mathsf{snd}, \mathsf{fst}\rangle \circ \langle g, \mathsf{id}\rangle && \\
&= \mu \circ T\hat{\tau} \circ \tau \circ (f \times \mathsf{id}) \circ \langle \mathsf{id}, g\rangle && \\
&= \mu \circ T\hat{\tau} \circ T(f \times \mathsf{id}) \circ \tau \circ \langle \mathsf{id}, g\rangle && \text{// naturality of } \tau \\
&= \mu \circ T(\hat{\tau} \circ (f \times \mathsf{id})) \circ \tau \circ \langle \mathsf{id}, g\rangle && \text{// functoriality of } T \\
&= (\hat{\tau} \circ (f \times \mathsf{id}))^\star \circ \tau \circ \langle \mathsf{id}, g\rangle. && \text{// since } (-)^\star = \mu \circ T
\end{aligned}
$$

Further important properties:

- *copyability:* $\mathsf{do}\, x \leftarrow p;\ \mathsf{do}\, y \leftarrow p;\ \mathsf{return}\langle x, y\rangle = \mathsf{do}\, x = p;\ \mathsf{return}\langle x, x\rangle$;
- *discardability:* $\mathsf{do}\, x \leftarrow p;\ \mathsf{return}\, \star = \mathsf{return}\, \star$.

Example. Powerset monad is commutative, but neither copyable, nor discardable.

Example (Probabilistic Computations)**.** The following is a probability distribution monad on *Set*:

- $DX = \{d \colon X \to [0,1] \mid \sum d = 1\}$ (it follows that the set $\{x \mid d(x) \neq 0\}$ is countable);
- $(\eta x)(x) = 1$ and $(\eta x)(y) = 0$ if $x \neq y$ (*Dirac's distribution*);
- $(f \colon X \to DY)^\star(d \colon X \to [0,1])(y \in Y) = \sum_{x \in X} d(x) \cdot f(x)(y)$.

This monad is commutative and discardable, but not copyable.

2.2.3 Applicative Functors

Applicative functors are a light-weight alternative to (strong) monads. Mathematically, they are the same as *strong lax-monoidal functors*.

Definition (Lax-monoidal functors)**.** A functor $F \colon \mathcal{C} \to \mathcal{D}$ between two Cartesian categories is called *lax-monoidal* if it is equipped with the following structure:

- a morphism $\epsilon \colon 1 \to F1$,
- a family of natural in X and Y morphisms $\sigma \colon FX \times FY \to F(X \times Y)$,

such that the following diagrams commute:

$$\begin{tikzcd}
1 \times FX \arrow[r, "\epsilon\times\mathsf{id}"] \arrow[d, "\mathsf{snd}"'] & F1 \times FX \arrow[d, "\sigma"] \\
FX \arrow[r, "F\langle !,\mathsf{id}\rangle"] & F(1 \times X)
\end{tikzcd}
\qquad
\begin{tikzcd}
FX \times 1 \arrow[r, "\mathsf{id}\times\epsilon"] \arrow[d, "\mathsf{fst}"'] & FX \times F1 \arrow[d, "\sigma"] \\
FX \arrow[r, "F\langle \mathsf{id},!\rangle"] & F(X \times 1)
\end{tikzcd}$$

$$\begin{tikzcd}
(FX \times FY) \times FZ \arrow[r, "\mathsf{assoc}"] \arrow[d, "\sigma\times\mathsf{id}"'] & FX \times (FY \times FZ) \arrow[d, "\mathsf{id}\times\sigma"] \\
F(X \times Y) \times FZ \arrow[d, "\sigma"'] & FX \times F(Y \times Z) \arrow[d, "\sigma"] \\
F((X \times Y) \times Z) \arrow[r, "F\,\mathsf{assoc}"] & F(X \times (Y \times Z))
\end{tikzcd}$$

A strong lax-monoidal functor is a lax-monoidal functor $F\colon \mathcal{C} \to \mathcal{D}$, which is strong and for which the following diagram commutes:

$$\begin{tikzcd}
(A \times FB) \times (C \times FD) \arrow[r, "\cong"] \arrow[d, "\tau\times\tau"'] & (A \times C) \times (FB \times FD) \arrow[d, "\mathsf{id}\times\sigma"] \\
F(A \times B) \times F(C \times D) \arrow[d, "\sigma"'] & (A \times C) \times F(B \times D) \arrow[d, "\tau"] \\
F((A \times B) \times (C \times D)) \arrow[r, "\cong"] & F((A \times C) \times (B \times D))
\end{tikzcd}$$

where $\cong$ refers to obvious isomorphisms.

This (in presence of exponentials!) is equivalent to the notion of applicative functor in Haskell which requires $\mathsf{pure}\colon A \to FA$ and $\langle * \rangle\colon F(B^A) \times FA \to FB$, and which can be defined as follows:

$$\mathsf{pure} = \big(A \xrightarrow{\langle \mathsf{id},!\rangle} A \times 1 \xrightarrow{\mathsf{id}\times\epsilon} A \times F1 \xrightarrow{\tau} F(A \times 1) \xrightarrow{F\,\mathsf{fst}} FA\big)$$
$$\langle * \rangle = \big(F(B^A) \times FA \xrightarrow{\sigma} F(B^A \times A) \xrightarrow{F\,\mathsf{ev}} FB\big)$$

Every strong monad is canonically an applicative functor as follows:

$$\epsilon = \eta \qquad\qquad \sigma = \mu \circ (T\hat{\tau}) \circ \tau$$

Dually, we obtain an applicative functor by taking $\sigma = \mu \circ (T\tau) \circ \hat{\tau}$. Unless the monad is commutative, this applicative functor structure is properly different.

2.3 Algebras and CPS-Transormations

Definition (Monad Algebras)**.** An *(Eilenberg-Moore) algebra* for a monad T, or a T*-algebra* is a tuple $(A, a\colon TA \to A)$ satisfying the following conditions:

$$\begin{tikzcd}
A \arrow[r, "\eta_A"] \arrow[dr, equal] & TA \arrow[d, "a"] \\
& A
\end{tikzcd}
\qquad
\begin{tikzcd}
TTA \arrow[r, "Ta"] \arrow[d, "\mu_A"'] & TA \arrow[d, "a"] \\
TA \arrow[r, "a"] & A
\end{tikzcd}$$

We call the object A of a T-algebra $(A, a\colon TA \to A)$ the *carrier* of the latter and the morphism $a\colon TA \to A$ the corresponding *structure*. As expected, morphisms of T-algebras are those morphisms of carrier that preserve the structure:

$$\begin{tikzcd}
TA \arrow[r, "Th"] \arrow[d, "a"'] & TB \arrow[d, "b"] \\
A \arrow[r, "h"] & B
\end{tikzcd}$$

We thus a category of T-algebras, of the Eilenberg-Moore category of T.

Example (Pointed Sets)**.** Let T be the maybe-monad $TX = X + 1$. Then $(A, a\colon A + 1 \to A)$ is a T-algebra iff

$$\begin{tikzcd}
A \arrow[r, "\mathsf{inl}"] \arrow[dr, equal] & A+1 \arrow[d, "a"] \\
& A
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
(A+1)+1 \arrow[r, "a+1"] \arrow[d, "{[\mathsf{id},\mathsf{inr}]}"'] & A+1 \arrow[d, "a"] \\
A+1 \arrow[r, "a"] & A
\end{tikzcd}$$

The former diagram means precisely that a is of the form $[\mathsf{id}, p]$ for some $p\colon 1 \to A$ and the latter diagram commutes automatically. Therefore, to give a maybe-algebra over A is to give a morphism $1 \to A$, i.e. specify a *point* in A. A morphism of algebras $h\colon (A, a\colon A+1 \to A) \to (B, b\colon B + 1 \to B)$ is exactly a morphism $h\colon A \to B$ of the carriers that respects the points.

Example (Monoids)**.** Let TX be the list monad over *Set*: $TX = X^\star$. It can be shown that the category of list-algebras is isomorphic to the category of monoids, defined as follows:

- objects are monoids $(M, \odot\colon M \times M \to M, e \in M)$;
- morphisms from $(M, \odot, e)$ to $(M', \odot', e')$ are those maps $h\colon M \to M'$, which preserve the monoid structure: $h(a \odot b) = h(a) \odot' h(b)$, $h(e) = e'$.

Definition (Free Algebras)**.** A free T-algebra on an object $A \in |\mathcal{C}|$ is the tuple $(TA, \mu_A\colon TTA \to TA)$.

The axioms of T-algebras are automatics for free algebras.

Definition (Strong Monad Morphisms)**.** Given two monads S and T on the same category, a natural transformation $\alpha\colon S \to T$ is a monad morphism if

$$\begin{tikzcd}
X \arrow[r, "\eta_X"] \arrow[dr, "\eta_X"'] & SX \arrow[d, "\alpha_X"] \\
& TX
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
SSX \arrow[r, "\alpha_{SX}"] \arrow[d, "\mu_X"'] & TSX \arrow[r, "T\alpha_X"] & TTX \arrow[d, "\mu_X"] \\
SX \arrow[rr, "\alpha_X"] & & TX
\end{tikzcd}$$

A monad morphism between two strong monads is strong if it is a strong natural transformation.

Monad algebras, strong monad morphisms and continuations are connected in the following theorem.

Theorem 9 (Dubuc's Theorem[3][4])**.** Given a strong monad T, T-algebra structures over $(A, a\colon TA \to A)$ are in one-to-one correspondence with strong monad morphisms $\alpha\colon T \to (\text{–} \to A) \to A$ as follows:

- given $(A, a\colon TA \to A)$,

$$\alpha_X = \mathsf{curry}\Big(TX \times (X \to A) \xrightarrow{\cong} (X \to A) \times TX \xrightarrow{(T\,\mathsf{ev})\tau} TA \xrightarrow{a} A\Big);$$

- given $\alpha\colon T \to (\text{–} \to A) \to A$,

$$a = \Big(TA \xrightarrow{\langle \mathsf{id}, \mathsf{curry\,snd}\rangle} TA \times (A \to A) \xrightarrow{\mathsf{uncurry}\,\alpha} A\Big).$$

If A is a free T-algebra $A = TR$ then $\alpha(p\colon TX)(f\colon X \to TR) = f^\star(p)$. Moreover, $\alpha(p\colon TR)(\eta\colon R \to TR) = \eta^\star(p) = p$. This can be illustrated with a series of Haskell programs. The program over the list monad

```
ex1 :: [Int]
ex1 = do
  a <- return 2
  b <- return 2
  return $ a+b
```

forms a list `[4]`. We can reuse just the same code for the continuation monad:

```
ex2 :: Cont String Int
ex2 = do
  a <- return 2
  b <- return 2
  return $ a+b
```

However, since the result type is `String`, in the end we will need to convert from `Int` to `String`, e.g. with `runCont ex2 show`. In contrast to the list monad we now can "escape" from the computation:

```
ex3 :: Cont String Int
ex3 = do
    cont (\r -> "escape")
    a <- return 2
    b <- return 2
    return $ a+b
```

[3] Eduardo J Dubuc. "Enriched semantics-structure (meta) adjointness". In: *Rev. Union Math. Argentina* 25 (1970), pp. 5–26.

[4] simplified version

Now, if we start with the program

```
ex4 :: [Int]
ex4 = do
  a <- [1,2]
  b <- [1,2]
  return $ a + b
```

which yields [2,3,3,4], we can use the CPS-transform of the list monad to convert to the continuation monad:

```
i x = cont (\r -> x >>= r)

ex5 :: Cont [Int] Int
ex5 = do
  a <- i [1,2]
  b <- i [1,2]
  return $ a + b
```

Here [Int] is the free list-algebra on Int and i is the induced monad morphism. With r ⌟ unCont ex5 return we obtain [42] like in the original case of the list monad. But now we also can escape from the computation:

```
ex6 :: Cont [Int] Int
ex6 = do
  cont (\r -> [42])
  a <- i [1,2]
  b <- i [1,2]
  return $ a + b
```

The same can be achieved with the library function callCC :: MonadCont m => ((a -> m b) -> m a) -> m a (=*call with current continuation*):

```
ex7 :: Cont [Int] Int
ex7 = callCC $ \k -> do
  k 42
  a <- i [1,2]
  b <- i [1,2]
  return $ a + b
```

2.4 Free Objects and Adjoint Functors

Definition (Free Objects)**.** Given a functor $G\colon \mathcal{C} \to \mathcal{D}$, a *free $\mathcal{C}$-object on $X \in |\mathcal{D}|$* consists of an object $FX \in |\mathcal{C}|$ together with a *morphism* $\eta_X : X \to G(FX)$ in $\mathcal{D}$ such that for any other $Z \in |\mathcal{C}|$ and morphism $f\colon X \to GZ$ in $\mathcal{D}$, there exists a unique $f^\dagger\colon FX \to Z$ in $\mathcal{C}$ such that

$$\begin{tikzcd}
G(FX) \arrow[r, "Gf^\dagger"] & GZ \\
X \arrow[u, "\eta_X"] \arrow[ur, "f"'] &
\end{tikzcd}$$

We used the suggestive notation FX, since F can be seen as an operation that sends given objects to free objects. If all free objects exist, F moreover extends to a functor, defined as follows: given a morphism $f\colon X \to Y$ in $\mathcal{D}$, let $Ff = \eta_Y \circ f$, which is the unique morphism, for which the following diagram commutes:

$$\begin{tikzcd}
G(FX) \arrow[r, "G(\eta_Y \circ f)^\dagger"] & G(FY) \\
X \arrow[u, "\eta_X"] \arrow[r, "f"] & Y \arrow[u, "\eta_Y"']
\end{tikzcd}$$

It can be show that F satisfies the axioms of functors:

- $F(\mathsf{id}_X) = \mathsf{id}_{FX}$, because id_{FX} makes the same diagram commute as $F\mathsf{id}_X = \eta_X$:

$$\begin{tikzcd}
G(FX) \arrow[r, "G(\mathsf{id}_{FX})"] & G(FX) \\
X \arrow[u, "\eta_X"] \arrow[r, "\mathsf{id}_X"] & X \arrow[u, "\eta_X"']
\end{tikzcd}$$

- $F(g \circ f) = Fg \circ Ff$, because $Fg \circ Ff = (\eta_Z \circ g)^\dagger \circ (\eta_Y \circ f)^\dagger$ makes the same diagram commute as $F(g \circ f) = (\eta_Z \circ g \circ f)^\dagger$:

$$\begin{tikzcd}
G(FX) \arrow[r, "G(\eta_Y \circ f)^\dagger"] \arrow[rr, bend left, "G((\eta_Z \circ g)^\dagger \circ (\eta_Y \circ f)^\dagger)"] & G(FY) \arrow[r, "G(\eta_Z \circ g)^\dagger"] & G(FZ) \\
X \arrow[u, "\eta_X"] \arrow[r, "f"] & Y \arrow[u, "\eta_Y"'] \arrow[r, "g"] & Z \arrow[u, "\eta_Z"]
\end{tikzcd}$$

Example (Free Monoids)**.** Let $\mathcal{C}$ be the category of monoids over **Set** and let G be the obvious forgetful functor. Then $(X^\star, \eta\colon X \to GX^\star)$ is a free object on X and for every $f\colon X \to GZ$, $f^\dagger\colon X^\star \to Z$ is a unique extension of f to a monoid map from $X^\star$ to Z.

Example (Free Algebras)**.** Let $\mathcal{C}$ be the category of T-algebras over $\mathcal{D}$ and $G\colon \mathcal{C} \to \mathcal{D}$ a forgetful functor. Let $F\colon \mathcal{D} \to \mathcal{C}$ be the free T-algebra functor. Then $(FX, \eta_X : X \to GFX = TX)$ is the free object on X.

Example (Exponentials)**.** Exponentials in a category $\mathcal{C}$ with binary products, can be viewed as *co-free objects*, i.e. free objects in $\mathcal{C}^{\mathsf{op}}$. To obtain the definition of co-free object we just invert all arrows in the definition of free object. To obtain exponentials X^A in $\mathcal{C}$, we take $\mathcal{D} = \mathcal{C}$, $GX = X \times A$ and then take $FX = X^A$. The corresponding diagram is then

$$\begin{tikzcd}
X^A \times A \arrow[d, "\mathsf{ev}"'] & Z \times A \arrow[l, "(\mathsf{curry}\, f)\times \mathsf{id}"'] \arrow[dl, "f"] \\
X &
\end{tikzcd}$$

More precisely, an exponential consists of an object X^A and an evaluation morphism $\mathsf{ev}\colon X^A \times A \to X$, such that for every $f\colon Z \times A \to X$, there is a morphism $\mathsf{curry}\, f\colon Z \to X^A$, which is the unique morphism with the property that $f = \mathsf{ev} \circ (\mathsf{curry}\, f \times \mathsf{id})$.

Definition (Adjointness)**.** A functor $F\colon \mathcal{D} \to \mathcal{C}$ is a *left adjoint* of $G\colon \mathcal{C} \to \mathcal{D}$ if $\mathsf{Hom}(FX, Y) \cong \mathsf{Hom}(X, GY)$ naturally in X and Y. This is written as $F \dashv G$ or $G \vdash F$ and G is called a *right adjoint* to F.

Theorem 10. A functor $G\colon \mathcal{C} \to \mathcal{D}$ has a left adjoint $F\colon \mathcal{D} \to \mathcal{C}$ iff there exist free objects $(FX, \eta_X\colon X \to GFX)$ for every X:

- from an adjunction $\mathsf{Hom}(FX, Y) \cong \mathsf{Hom}(X, GY)$ we obtain a correspondence

$$(f\colon X \to GY) \mapsto (f^\dagger\colon FX \to Y)$$

such that $(\eta_X\colon X \to GFX)^\dagger = \mathsf{id}_{FX}$ for a suitable η_X;

- from free objects $(FX, \eta_X\colon X \to GFX)$, we obtain the maps

$$\begin{aligned}(f\colon FX \to Y) &\mapsto ((Gf)\eta\colon X \to GY),\\ (f\colon X \to GY) &\mapsto (f^\dagger\colon FX \to Y).\end{aligned}$$

Theorem 10 allows us to switch between two equivalent ways of defining categorical structures: by adjunctions or by free objects. The latter way is more fine grained, because we can speak about existence of *specific* free objects, while the adjoint formulation is only sensible when *all* free objects exist.

Example (Exponential)**.** Existence of exponentials now can be reformulated as $(-) \times A \dashv (-)^A$. Theorem 10 shows that this definition is equivalent to the definition via free objects.

By Theorem 10, we now see that $F \dashv G$ for F being the free T-algebra functor and G being the corresponding forgetful functor. This is called the *Eilenberg-Moore adjunction*. Because of Theorem 10, it is easy to see that we could just as well consider the category of free T-algebras instead of the category of all algebras. The resulting adjunction is called the *Kleisli adjunction*. The reason for it is the following

Proposition. The Kleisli category of a monad is isomorphic to the category of all free algebras of that monad. The relevant isomorphism is defined as follows:

- *(from Kleisli for free algebras):*

$$X \mapsto (TX, \mu_A), \qquad (f\colon X \to TY) \mapsto (f^\star\; TX \to TY);$$

- *(from free algebras to Kleisli):*

$$(TX, \mu_A) \mapsto X \qquad (f\colon (TX, \mu_X) \to (TY, \mu_Y)) \mapsto (f\eta\; X \to TY)$$

Bibliography

Barendregt, Hendrik. *The Lambda calculus: Its syntax and semantics.* Amsterdam: North-Holland, 1984.

Dubuc, Eduardo J. "Enriched semantics-structure (meta) adjointness". In: *Rev. Union Math. Argentina* 25 (1970), pp. 5–26.

Kock, Anders. "Strong Functors and Monoidal Monads". In: *Archiv der Mathematik* 23.1 (1972), pp. 113–120.