Category Theory and Computational Complexity

Marco Larrea Octavio Zapata

December 27, 2014

1 Introduction

1.1 Computability

Here some rudiments of the computability theory and complexity theory.

1.1.1 Turing machines

A Turing machine is a particular assembly and mode of operation of the following components: a finite set of symbols S which contains element $\perp \in S$ called the "blank" symbol; a subset $\Sigma \subseteq S \setminus \{\perp\}$ called the *input alphabet*; a finite set of states Q which contains an initial state $q_0 \in Q$; a partial function $\delta : Q \times S \to Q \times S \times \{-1, 0, 1\}$ called the transition function.

Computation is performed by a read-write head on symbols over a tape divided into cells, each cell carries one symbol from S. The tape is infinite to the right; its content is a sequence $\sigma = \sigma_0, \sigma_1, \ldots$ where $\sigma_i \in \Sigma$. The head in position $p \in \mathbb{N}$ can read symbol s_p and write another symbol in its place. The configuration of a Turing machine is a triple $\langle \sigma; p; q \rangle$ where $q \in Q$.

At each step the machine computes $\delta(q, s_p) = (q', s', \Delta p)$, which determines its new configuration $\langle s_0, \ldots, s_{p-1}, s', s_{p+1}, \ldots; p + \Delta p; q' \rangle$ after the transition. A Turing machine *halts* whenever $\delta(q, s_p)$ is undefined, or $p + \Delta p < 0$. Otherwise, it may never stops.

Inputs and outputs are strings over Σ. Initially the tape is filled with $\sigma \in \Sigma^*$ and padded with blanks; the head is at the left end so the initial configuration of the machine is $\gamma_0 := \langle \sigma \perp \perp \ldots; 0; q_0 \rangle$. If the machine eventually halts, the output is the string written on the tape. In general, a computation can be seen as a sequence of configurations

$$\langle \sigma \perp \perp \ldots; 0; q_0 \rangle \to \langle \sigma'; p; q \rangle \to \cdots$$

which we will denote by $\gamma_0 \vdash \gamma_1 \vdash \cdots$.

Every Turing machine M computes a partial function $\Phi_M : \Sigma^* \to \Sigma^*$. By definition, $\Phi_M(\sigma)$ is the output string for input σ. The value $\Phi_M(\sigma)$ is undefined if the computation never terminates.

A partial function $f : \Sigma^* \to \Sigma^*$ is *computable* if there exists a Turing machine M such that $\Phi_M = f$. In this case we say that f is computed by M.

A predicate is a property that can be true or false. Predicates whose domain of discourse is the set Σ^* can be identified with languages $\{x : P(x)\} \subseteq \Sigma^*$ over Σ. Formally, a predicate is a function $P : \Sigma^* \to \{0, 1\}$ which is said to be decidable if there exists a Turing machine that computes it.

A Turing machine M works in time $T(n)$ if it performs at most $T(n)$ steps to computes $\Phi_M(\sigma)$ for any $\sigma \in \Sigma^*$ such that $|\sigma| = n$. Analogously, M works in space $s(n)$ if $|\Phi_M(\sigma)| \leq s(n)$ for any $\sigma \in \Sigma^*$ such that $|\sigma| = n$.

A nondeterministic Turing machine is a Turing machine which have a multivalued transition function, i.e. the function δ is not an injection.

For some purpose it may be convenient to consider multitape Turing machines that have a finite number of tapes. Each tape has a head that can read and write symbols on it. However, two special tapes are distinguished: an input read-only tape, and an output write-only tape. The remaining tapes are called the work tapes.

1.1.2 Complexity classes

Let f and g be two functions. For sufficiently large n, if $|f(n)| \leq k \cdot g(n)$ for some $k > 0$, we write $f(n) \in O(g(n))$; if $f(n) \geq k \cdot g(n)$ for some $k > 0$, we write $f(n) \in \Omega(g(n))$. If, particularly $f(n) \leq c \cdot n^k$ for some c, k constants, we write $f(n) = \mathsf{poly}(n)$.

A function F on Σ^* is computable in polynomial time if there exists a Turing machine that computes it in time $T(n) = \mathsf{poly}(n)$, for every $\sigma \in \Sigma^*$ such that $|\sigma| = n$. If F is a predicate, we say that it is decidable in polynomial time.

We define the complexity class P as the class of all functions computable in polynomial time, or all predicates decidable in polynomial time. Notice that $F \in \mathsf{P}$ implies $|F(x)| = \mathsf{poly}(|x|)$.

A function (predicate) F on Σ^* is computable (decidable) in polynomial space if there exists a Turing machine that computes F and runs in space $s(n) = \mathsf{poly}(n)$, for every $\sigma \in \Sigma^*$ such that $|\sigma| = n$.

The class of all functions (predicates) computable (decidable) in polynomial space is called PSPACE. Notice from the definitions that clearly $\mathsf{P} \subseteq \mathsf{PSPACE}$.

Complexity class NP is defined as the class of predicates decidable in polynomial time by nondeterministic Turing machines. Notice that NP is defined only for predicates. The class coNP is defined as the class of predicates whose complements are in the NP class.

Let $\Sigma_0^\mathsf{P} := \Pi_0^\mathsf{P} := \mathsf{P}$. For $i \geq 0$, define $\Sigma_{i+1}^\mathsf{P} := \mathsf{NP}^{\Sigma_i^\mathsf{P}}$ and $\Pi_{i+1}^\mathsf{P} := \mathsf{coNP}^{\Pi_i^\mathsf{P}}$ where A^B is the class of languages accepted by a Turing machine that computes languages in A, augmented by a particular machine that computes languages in B. Notice that $\Sigma_1^\mathsf{P} = \mathsf{NP}$.

The polynomial hierarchy $\Sigma_*\mathsf{P}$ is defined as

$$\Sigma_*\mathsf{P} \;:=\; \bigcup_{i\in\mathbb{N}} \Sigma_i^\mathsf{P} \;:=\; \bigcup_{i\in\mathbb{N}} \Pi_i^\mathsf{P}.$$

Claim 1.1. *It follows from the definitions that*

$$\mathsf{P} \subseteq \{\mathsf{NP}, \mathsf{coNP}\} \subseteq \Sigma_*\mathsf{P} \subseteq \mathsf{PSPACE}.$$

1.2 Descriptive complexity

Let L be a logic, C a complexity class and K a class of finite structures. We say that L captures C on K if for every vocabulary τ and every property $\mathcal{P} \in \mathsf{K}(\tau)$

$$\mathcal{P} \text{ is } \mathsf{L}\text{-definable} \;\Leftrightarrow\; L(\mathcal{P}) \in \mathsf{C}.$$

This fact is denoted as $\mathsf{L} = \mathsf{C}$.

1.3 First-order dependence

A first-order dependence logic D is a class which consists of all D-definable properties where $\mathsf{D} := (\mathsf{FO} + \mu.\bar{t})$ and $\mu.\bar{t}$ denotes that term $t_{|\bar{t}|}$ is functionally dependent on t_i for all $i \leq |\bar{t}|$. The model class FO is as always defined as the class of models of all first-order sentences (i.e. $\mathsf{FO} := \{S : (\exists\tau)(\exists\varphi \in L(\tau))\ S = \mathsf{Mod}(\varphi)\}$ where $L(\tau)$ is a first-order language of type τ) and $\mu.\bar{t}$ is interpreted as a recursively generated tuple of terms which we naturally identify with the set $[|\bar{t}|] := \{1, 2, \ldots, |\bar{t}|\}$. D sentences are capable to characterise variable dependence and in general they are proven to be as expressive as the sentences of the second order existential $\mathsf{SO}\exists$ model class [Vää07]. The intuitionistic dependence version ID has the same expressive power as full SO [Vää07]. It is a fact that MID-model checking is PSPACE-complete [Vää07] where MID is the intuitionistic implication fragment of the modal dependence logic MD which contains at least two modifiers. Hence, $(\mathsf{FO} + \mu.\bar{t}) = \mathsf{NP}, \mathsf{ID} = \Sigma_*\mathsf{P}$ and $\mathsf{MID} = \mathsf{PSPACE}$. On the other hand, $\mathsf{PSPACE} = \mathsf{IP} = \mathsf{QIP}$ [JJUW11], and so $\mathsf{MID} = \mathsf{QIP}$ which is the quantum version of the interactive polytime class IP.

We shall try to cook up a purely algebraic definition for the class of structures MID and extend such algebraic logic in order to capture other quantum and classical complexity classes.

1.4 Ultraproduct

In this section every vocabulary τ will be referred as a similarity type or simply as a type.

The model class FO is, as always, defined as the class of models of all first-order sentences, i.e. $\mathsf{FO} := \{S : (\exists\tau)(\exists\varphi \in L(\tau))\ S = \mathsf{Mod}(\varphi)\}$ where $L(\tau)$ is a first-order language of type τ.

Ehrenfeucht-Fraïssé games characterise the expressive power of logical languages [Imm]. Every Ehrenfeucht-Fraïssé game is an ultraproduct [Vää11], a back-and-forth method for showing isomorphism between countably infinite structures, but only defined for finite structures in finite model theory. If F is an ultrafilter (i.e. $F \subseteq 2^{\mathbb{N}}$ and $\forall X \subseteq \mathbb{N}(X \notin F \Leftrightarrow \mathbb{N} \setminus X \in F)$ holds) then the reduced product $\prod_i M_i/F$ is an ultraproduct of the sets M_i, $i \in I$. Recall that

$$f \sim g \Leftrightarrow \{i \in I : f(i) = g(i)\} \in F$$

for all infinite sequences $f, g \in \prod_i M_i$ and any index set I, is the relation which induces the equivalence classes that conform the ultraproduct

$$\prod_i M_i/F = \{[f] : f \in \prod_i M_i\}.$$

This mathematical tool (the ultraproduct) is widely important because of results such as the following, from which proof we will delayed for the moment.

Lemma 1.2 (Loś Lemma)**.** *If F is an ultrafilter and φ a first-order formula, then the ultraproduct of models of φ indexed by any index set $I \in F$ is a model of φ, i.e.*

$$(\prod_i A_i/F, \alpha) \models \varphi \Leftrightarrow \{i \in I : (A_i, \alpha_i) \models \varphi\} \in F.$$

1.4.1 Ultrafilters

2 Categorical Semantics of the Lambda Calculus

The $\lambda-$calculus is an abstraction of the theory of functions, in the same way group theory is an abstraction of the theory of symmetries. There are two basic operation on function we would like to formalize, *application* and *abstraction*.

Application refers to the operation performed by a function on a given term or expression. For example, if *double* is the function that multiplies by two, then for any given natural number n, we can apply *double* to n to form the new natural number $double(n) = 2n$. Note that in order to be consistent one should define the type of arguments a function can take, for instance, it makes no sense to apply *double* to a string "*string*" of characters.

Abstraction is the operation of introducing new functions. Given a term t which (possibly) depends on a variable x, we can form a new function by abstracting the variable x from the term t in such a way that the application of this function on a term u is given by substituting in t the variable x by u. So for example, if we have the term $t = x*2$ which depends on x, we form the function $\lambda x.t$ which extensionally is the same as the function *double* from above, that is $\lambda x.t(n) = double(n) = 2n$ for all natural number n.

The *simply-typed lambda calculus* is a form of type theory that interpretes the λ-calculus. Types are used in order to improve the consistency of the originally untyped theory.

The first step to define the simply-typed lambda calculus is to fix a set β whose elements we name *basic types* or *atomic types*. We express the fact that an object is a *type* by the judgment:

$$A \ type$$

We want every element of β to be a type, for this we introduce an *axiom* which is a special kind of *deduction rule* for which there are no assumptions. So for each $A \in \beta$ we have the rule:

$$\frac{}{A \ type}$$

which is read "A is a type". We'll also want to have a special type with only one term which we shall name the *unit type*:

$$\frac{}{1 \ type}$$

There are two introduction rules for types, these rules tell us how to construct new types from old ones. There is the introduction rule for *product types*:

$$\frac{A\ type \qquad B\ type}{A \times B\ type}$$

and the introduction rule for *function types*:

$$\frac{A\ type \qquad B\ type}{A \to B\ type}$$

Therefore the set of all types of the simply-typed lambda calculus is recursively generated from the set of basic types by applying the introduction rules of products and functions.

Now the set of types is defined we would like to define in a similar way the set of terms. As before we fix a set of *constant terms* or just *constants*. We also asume there are countable many variables (or as many as we might need), we'll name the variables $x, y, z, \ldots$. Just as types we will recursively generate the set of terms as follows:

$$t := [variables] \mid [constants] \mid * \mid < t, t' > \mid \pi_1 t \mid \pi_2 t \mid t(t') \mid \lambda x.t$$

A term of the form $t(t')$ is called an *application* and one of the form $\lambda x.t$ is called a *lambda abstraction*.

There are two ways a variable x can appear in a given term t. We say that x is *bound* in t if it appears in the scope of a lambda abstraction, i.e. if inside of t there is a substring of the form $\lambda x.t'$ for some t'. *Free* variables are those that are not bound.

A very important syntactic operation of terms is *substitution*. Given a term t (such that a variable x may ocurre freely in it), for any term u, the new term

$$t[u/x]$$

is obtained from t by substituting the free occurrences of the variable x by u. For example, consider the term:

$$t =< \lambda x. < x, y >, z >$$

In t we have that the variables y and z are free while x is bound. Let $u =< a, \lambda w.w >$, we can substitute in t the variable y for u to form the next term:

$$t[u/y] =< \lambda x. < x, < a, \lambda w.w >>, z >$$

If we try to substitute the variable x for u we would obtain the term t again since x is bound, i.e. $t[u/x] = t$.

A *context* is a finite sequence of typed variables, that is, if $x_1, x_2, \ldots, x_n$ are distinct variables and $A_1, A_2, \ldots, A_n$ are types, the sequence

$$x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n$$

is a context, the empty sequence is a valid context. We denote contexts with capital greek letters $\Delta, \Gamma, \ldots$

A *typing judgment* is of the form:

$$\Gamma \vdash t : A$$

where Γ is a context, t a term and A a type. We read the above judgement as "In context Γ the term t is of type A". Intuitively what we mean by this judgement is that the free variables of t must ocurre in Γ.

To form new typing judgements we need to follow some specified rules. We will now enunciate the rules of the simply-typed lambda calculus.

- Each constant term has a unique determined type, i.e. for each constant a there is a unique type A such that for every context Γ we have that:

$$\frac{}{\Gamma \vdash a : A}$$

- The type of a variable is determined by the context, i.e. if $x_i : A_i$ ocurre in a context Γ we have that:

$$\frac{}{\Gamma \vdash x_i : A_i}$$

- The term $*$ has type 1 in every context:

$$\frac{}{\Gamma \vdash * : 1}$$

- The context of a given judgement can be extended. This rule is known as *weakening*:

$$\frac{\Gamma \vdash t : A}{\Gamma, \Delta \vdash t : A}$$

  where the context Γ, Δ is given by the union $\Gamma \cup \Delta$.

- Rules for product types:

$$\frac{\Gamma \vdash u : A \qquad \Gamma \vdash t : B}{\Gamma \vdash < u, t > : A \times B} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_1 t : A}$$

$$\frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_2 t : B}$$

- Rules for function types:

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t(u) : B} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}$$

  where $\Gamma, x : A$ is the context $\Gamma \cup \{x : A\}$.

Apart from the typing judgment there is another kind of syntactic judgement, the *equality judgement*, this specifies when two terms of the same type (under the same context) are equal. It's written as follows:

$$\Gamma \vdash t = u : A$$

Equality must at least be an equivalence relation, so we have the following rules:

$$\frac{}{\Gamma \vdash t = t : A} \qquad \frac{\Gamma \vdash t = u : A}{\Gamma \vdash u = t : A} \qquad \frac{\Gamma \vdash t = u : A \qquad \Gamma \vdash u = v : A}{\Gamma \vdash t = v : A}$$

The equality judgement must behave well with the rules for typing judgements, this means that for each of the rules seen before, there is an appropriate coherence rule for equality judgement. For example, for the application rule of functions, we have the corresponding rule:

$$\frac{\Gamma \vdash f = g : A \to B \qquad \Gamma \vdash u = v : A}{\Gamma \vdash f(u) = g(v) : B}$$

This is true for all the rules previously stated: weakening rule, product and function rules and unit rule.

Finally there are three important rules that are meant to formalize the important notions of function application and abstraction.

- The *α-conversion* rule formalizes the notion that bound variables in a given term can be interchanged without altering the meaning of the term:

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t = \lambda y.t[y/x] : A \to B}\ (\alpha\text{-}conv)$$

  where y is a variable that does not ocurre freely in t.

- The *β-reduction* rules tells us that if we first abstract a variable x from a term t and then apply the resulting function to another term u, the result must be the same as syntactically substituting u for x in the original term t:

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash (\lambda x.t)(u) = t[u/x] : B}\ (\beta\text{-}redu)$$

- The η-*reduction* formalizes the notion of function extensionality, i.e. that two function that have the same output on each term are equal:

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash \lambda x.f(x) = f : A \to B}\ (\eta\text{-}redu)$$

The above collection of data characterise the simply-typed lambda calculus, so let's summarize this in a definition.

Definition 2.1. A *simply-typed lambda calculus* is given by a set of Basic Types and a set of Constants subject to the rules given above.

References

[Ala13] Jesse Alama. The lambda calculus. http://plato.stanford.edu/archives/fall2013/entries/lambda-calculus/, 2013.

[APW13] Steve Awodey, Alvaro Pelayo, and Michael A. Warren. Univalence axiom in homotopy type theory. *Notices of the AMS*, 60(9):1164–1167, 2013.

[CF58] Haskell B Curry and Robert Feys. *Combinatory Logic, Studies in Logic and the Foundations of Mathematics*, volume 1. North-Holland, Amsterdam, 1958.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, Jun. 1940.

[Coq13] Thierry Coquand. Coq. coq.infra.fr, 2013.

[How95] W. A. Howard. The formuæ-as-types notion of construction. In *The Curry-Howard Isomorphism*. Academia, 1995.

[Imm] Neil Immerman. Descriptive complexity: a logician's approach to computation.

[JJUW11] Rahul Jain, Zhengfeng Ji, Sarvagya Upadhyay, and John Watrous. Qip = pspace. *J. ACM*, 58(6):30:1–30:27, December 2011.

[KLV12] Chris Kapulkin, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. The simplicial model of univalent foundations. *arXiv preprint*, Nov. 2012.

[ML84] Per Martin-Löf. An intuitionistic theory of types: Predicative part. *The Journal of Symbolic Logic*, 49(1):311–313, Mar. 1984.

[Nor13] Ulf Norell. Agda. wiki.portal.chalmers.se/agda/pmwiki.php, 2013.

[Sch13] Urs Schreiber. Infinity groupoid. http://ncatlab.org/nlab/show/infinity-groupoid, 2013.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[Vää07] J. Väänänen. *Dependence Logic: A New Approach to Independence Friendly Logic*. London Mathematical Society Student Texts. Cambridge University Press, 2007.

[Vää11] J. Väänänen. *Models and Games*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2011.

[Voe06] Vladimir Voevodsky. Foundations of mathematics and homotopy theory. http://video.ias.edu/node/68, Mar. 2006.